BASIC ADVENTURE and STRATEGY GAME DESIGN for THE TRS-80®

A step-by-step guide to designing elaborate programs in Basic, including complete program listings for the original adventure "Space Derelict!" and Five-Player Draw Poker

JIM MENICK

Basic ADVENTURE and STRATEGY Game Design for the TRS-80

Basic ADVENTURE and STRATEGY Game Design for the TRS-80

JIM MENICK

Facts On File Publications
New York, New York • Bicester, England

Basic Adventure and Strategy
Game Design for the TRS-80

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Designed by Giorgetta Bell McRee

CONTENTS

ACKNOWLEDGMENTS

The program listings in this book were made from photostats of the actual programs, double-struck on a trusty Epson MX-80 dot-matrix printer. The design specifications of this book necessitated some infelicitous line breaks, but balanced against this is the nice touch of new commands breaking at each colon. This was done by transferring the programs to an Apple II$^+$ and using the DOUBLE-TAKE utility program written by Mark Simonsen and published by Beagle Brothers for the Apple.

My friend Peter Schneider, who eats adventures for breakfast, was most helpful in finding fleas in ''Space Derelict!'' And I must thank Marvin, Walter, and David/Betsy, who took me in many years ago as a player in their monthly poker game and whose styles of play I have so shamelessly recreated here. Gerry, on the other hand, I brought into the poker game myself; I think he's financially ahead after all these years, so maybe he should thank *me*. But that notwithstanding, Gerry was also the editor of this book (poker buddies tend to stick together), and he must be credited with making sure I didn't get carried away by my own words. Further, it was Gerry who managed to get me my Model 4 in the first place, an extraordinary act beyond the call of normal editing.

For buying our original personal computer and then watching me slowly but steadily make it my own, for advice on business programming and system design, and most of all for extreme, enduring patience as I plugged away at this book, I thank my wife, Liz Pereira. And finally, for displaying all the marvelous wisdom of a three-year-old as she so kindly allowed me to work on this book during what occasionally was her time, I thank my daughter, Kate.

Basic ADVENTURE and STRATEGY Game Design for the TRS-80

INTRODUCTION

The TRS-80 Model 4 was my second computer. Even though I felt I knew something about the subject, I found myself more than a little daunted when I unpacked this impressive-looking machine and set it up on the table. The *Model 4 Disk System Owner's Manual* looked confusing to say the least, and the "Introduction to Your Disk System" pamphlet immediately made it clear that I'd also have to go out and buy *Getting Started with TRS-80 Basic* in addition to what I already had, or else getting started would always be beyond me. Needless to say, when I booted the TRSDOS disk and read the directory I was presented with nothing but meaningless gibberish. But I was expecting that—my original Apple II System Disk had been no clearer. I remembered back to the first, most important lesson I learned about computers: When it comes to learning about computers, you're on your own. In those early days I dove into the computer literature pool eagerly and thus was eventually able to formulate Menick's Law of Computer Literature: All computer books are either too easy or too hard. Most of the books tended to fall into the latter category (and were usually abominably written in the bargain), and I felt that if I could understand these books I wouldn't have needed them in the first place. As for the easy ones, they simply recapitulated the information in the manuals without addressing any grand purpose, on the assumption that if it didn't make sense the first time perhaps another run-through might make some of it stick.

There is another category of book that does have its attraction: collections of program listings. I am not inclined to sit for endless hours going blind trying to copy some third-graders' program for "Hammurabi," but it *is* interesting to see how others have solved problems that you have encountered yourself. There is probably more to be learned from studying a well-designed program than from any other approach to programming. Seeing how others have handled things, seeing the thought processes in action, is worth a hundred lectures on the theory of subroutines. It's especially interesting to note problem-solving failures where you can find an even better way to do the job than the original programmer.

While we're on the subject of other people's programs, I might as well air a pet peeve of mine: machine-language programs masquerading as Basic programs, which seem to be the mainstay of some books and magazines. I find nothing more frustrating than sitting down to a long session with POKE this and POKE that, CALL home, etc., and being expected to make heads or tails of it. And I think this is something a lot of hobbyists share with me. POKE commands are entirely unintelligible unless you know what business is taken care of at that machine address (if you don't know what I'm talking about now, don't worry; if you did know, you'd be sure to agree with me). And I have yet to see any writer make more than the most superficial stab at explaining what he is doing when he gets around to POKEing things. Since the manuals deal with POKE in only the most casual of references, I've always wondered why writers think all readers are equally adept at it.

In any case, the programs in this book will be sticking to more familiar Basic ground. It doesn't take long in the TRS world to realize that there are as many different versions of DOS and TRS Basic as there are Radio Shack stores in the suburbs. We're going to use the TRSDOS 6.0.0. and the accompanying version of disk Basic, the lingua franca of the Model 4. If you've got some slightly different configuration you still should be able to get by with what we're doing here, since we'll be dealing mostly with Basic itself rather than the tricks of the individual machines.

The approach of this book is straightforward. Part One is devoted to adventure games, because they are easier to design than the strategy games that comprise Part Two. Adventures can be designed, for the most part, without many subroutines; adventures are fairly straightforward sorts of programs, and the few subroutines are simply of a bookkeeping nature. Strategy games, on the other hand, consist of one subroutine after the other, linked together in a logical progression but nonetheless easily long enough to get lost in. Here I will try to guide you through your own jungles.

Throughout the book the direction will be from the general to the specific. Concepts will be explained as they pertain to Basic and games in general and then will be put to work in one of the two complete game programs given in the book. In that way there's the double reinforcement of both getting a definition and actually seeing how something works in action. And I'm assuming no prior knowledge on your part, aside from the perfunctory reading of the manuals that came with the machine in the first place, but if you already know your way around, you should still be able to find some helpful new ideas here. I'll assume you know what a PRINT statement is and how it works, and not much else, but I'm not going to beat you over the head with it either. If you're not passingly acquainted with the elementary commands you should go back to the manuals one more time to refresh yourself whenever a command comes up in this book that you don't feel 100 percent sure of. In addition, in the back of the book there's an elaborately chatty Glossary of all the terms we'll be using. Among the reference manuals, the Glossary, and the text itself you should get a good, solid feel for all of the commands and be able to use them in your own programs as the need arises.

As you read this book you might be able to see other ways of doing the same things I'm doing in my programs. There is no one way to solve a computer problem, and if you feel you can improve on my designs, feel free to do so. The point of this book is to provide comprehensive, reusable structures for game programs and to throw in a few tips that are otherwise hard to come by. I make no claims that my ways are the only ways or even the absolutely best ways; my point is that they work, and they are as simple as I can make them. Beyond that, you're on your own.

PART 1
ADVENTURE GAMES

1

PLANNING YOUR UNIVERSE

Simply put, an adventure game is a puzzle. The player either has to collect all the treasures hidden throughout the game universe or else has to manipulate successfully through the game universe to perform some specific task, such as saving the Earth from certain destruction. The player sifts through a variety of clues and situations, puts them together in the correct way, and thereby solves the puzzle. The measure of a good adventure game is its complications: the more twists and turns, the more meat for the player to chew on. A game with six rooms, for instance, is almost by definition less interesting than a game with 12 rooms. By the same token, a game with six complications is certainly less interesting than a game with 12 complications. In the long run, creating a good adventure game boils down to creating good complications.

If a game has 30 rooms and each room has a treasure in clear sight, and all the player has to do is plot his way through the rooms and collect the treasures, the only complication is finding the rooms. A simple complication you could add to this is one of inventory, limiting the player to carrying five items at one time. Now, at least, even though the treasures are easily accessible, the player has to do a lot of running around to carry everything. But there still isn't much mystery to the undertaking. The next complication might be to booby-trap some of the treasures. The unthinking player, bending down nonchalantly to pick up the diamond necklace, finds himself bitten by a cobra who pops out from under it. The player dies immediately and in his next incarnation has to figure out a way to remove the cobra before he can remove the necklace. Needless to say, somewhere in your game you have planted either a mongoose or a snake trap or something else that will do the job. Now we're really getting complicated. Add another twist, hide a good proportion of the treasures, and you're really cooking. For that matter, why not hide some of the rooms as well? Now our player not only is limited in the amount he can carry, he also finds it difficult to map the universe and collect the treasures. Depending on how complicated you make your complications (and that is the crux of the art of adventure designing), either the player is now in for hours of enjoyment trying to figure out what your universe is all about, or else he'll breeze right through it because it was too simple, or else he'll *never* figure it out and he'll curse you forever.

There isn't much difference between a treasure-gathering adventure and a situational adventure insofar as the individual complications are concerned. In a situational adventure the player still walks around through the game universe trying to get past individual booby traps and trying to find the hidden pieces; the only real differences are that a situational adventure should be tighter thematically and that instead of picking up treasure the player moves through plateaus of experience until he finally performs the task he has set out to do.

Two popular early games by Scott Adams clearly point up the differences, and these games are highly recommended to anyone who wants to see what a good game designer can come up with. ''Adventureland'' is a treasure-gathering game. Thirteen treasures are scattered around; some are easy pickin's, others are well hidden, and almost all of them require a trick to pry them loose. There is no real theme to ''Adventureland,'' although there are minithemes to most of the treasures; for instance, there is an ax with the magic word ''Bunyan'' printed on it, and there is a sign that says ''Paul's Place.'' More I cannot say without giving away too much of the fun. In contrast, the Adams adventure ''Mission Impossible'' has no treasures. At the beginning of the game the player is told that a saboteur is planning to blow up a nuclear reactor; the player's mission (''if you decide to accept it'') is to defuse the bomb. To do this the player has to trick his way past a number of locked doors, and past those doors he must manipulate certain objects in such a way as not to blow himself up as he proceeds. The game is won when the bomb in the reactor is deactivated.

SUBJECTS FOR ADVENTURES

The first decision you have to make in adventure designing is easy: treasure hunt or situational game. There's no dramatic difference in the programming, so it boils down mostly to which type of game you prefer playing yourself. The real problem you face at the outset is the idea for a game. It's all well and good to want to design adventures, but your adventures ought to be original and interesting. If you're into adventure gaming you've probably already realized that just about every possible subject seems to have been covered. There are pirate games, haunted-castle games, deserted-island games, amusement-park games; you name it. But ''Mission Impossible,'' for instance, by taking a very specific subject, managed to carve out a special niche for itself, and my recommendation is that you try to carve out similar unique niches. The first adventure game I designed was based on the film *Casablanca*. The premise was simple, and a detailed knowledge of the film was not necessary. The player took the part of Rick

(Humphrey Bogart). The object of the game was to find Victor Laszlo, the French Resistance man, and get him safely to the airport. The problem the player faced was the ubiquitous Nazi presence. In addition to Victor Laszlo and his wife (Ingrid Bergman in the film), the player had to find and manipulate a gun, a passport, letters of transit, a boat that wouldn't start, and various other items and people. An avaricious beggar in possession of a well-equipped secret hiding place, for instance, was purely of my own invention and never dreamed of by the filmmakers.

Since my personal tastes are literary and cinematic, those sources tend to provide me with the most appealing game ideas. If you're of a mind to sell your games eventually, you must of course keep in mind copyright laws and steer clear of protected material (or else pay the proprietor a usage fee). Some ideas that come off the top of my head and are free of copyright hassles would be any number of stories by Edgar Allan Poe ("The Pit and the Pendulum," for instance, sounds like a great game, whatever you might put into it). Jules Verne is similarly ripe (*Around the World in 80 Days*—there's enough material there for 80 disks!). If you're really into it you might even try Shakespeare. (*The Tempest* sounds pretty good, Ariel and Caliban, shipwrecks and magic, a desert island. Or *Macbeth,* with murders and ghosts and witches). When you start thinking about it, the possibilities are endless and fascinating; all you have to do is start thinking in the right direction.

The inspiration for "Space Derelict!" is my own, although since its development I've discovered other, similar premises. My real object when I started working on it was to legitimize the fact that the player is sitting at a computer terminal. I wanted to design a game that was realistic right at the start, so I began working from the concept of player-at-computer and branched out from there. Eventually I decided that the player would be at Mission Control and that his computer would be hooked up to a robot at the other end. The player would manipulate the robot, communicating to it through his computer terminal, and the robot's responses would be communicated back to him on the monitor. The next step was coming up with a logical scenario for such a situation, and I decided that the robot would be a sort of space probe, tirelessly traveling the solar system as an extension tool for human beings. In "Space Derelict!" our robot comes upon what is apparently a derelict ship floating in space and proceeds to examine it under the guidance of Mission Control. Keeping with the concept, I tried to make the robot sound like a robot as much as possible, since I think such twists give an adventure a little kick of their own.

The main thing to keep in mind when working over game ideas is originality. The players who try your game will, most likely, be familiar with other adventure games. To keep their interest you should present them with a new universe, a new world to roam around in, an adventure different from the other adventures they have played. Your own reading, or your own favorite films, are the best places to look for inspiration as a starting point for your creativity.

HOW TO PLAN SCENARIOS

After you've decided the general subject of your adventure, the next step is to plan it out in its entirety. The first thing to do is draw a map, which is not surprising since this is the first thing the players will do. At the same time you also outline the design of the play; that is, you plot out the adventure as the player will go through it, trying to get a handle on just what he will come up against. At the same time you figure out all the complications and how the player will solve them. At the same time you tally up all the objects the player will be manipulating and decide where to plant them. At the same time you go crazy trying to do all these things at the same time, and you begin to realize why a good adventure is a fairly rare commodity.

The best way to handle design is first to figure out the main object of the game and then to break the game down into smaller phases. If your game is a treasure hunt with no overall thematic progression, then you're into phases right at the outset. If your game is situational, you simply backtrack from the big complication (defusing the bomb that will blow up the galaxy or whatever) back to the little ones.

After having decided in the situational game "Space Derelict!" to use the concept of a robot that encounters a derelict space ship, I had to figure out the so-what of it all. I wanted to create a truly alien environment where objects were not necessarily what they appeared to be, or were sometimes simply incomprehensible from the human (or robotic) perspective. To make it all come together I needed an ultimate objective other than just roaming around the ship. So I lit on the idea that this apparently derelict ship was actually a bomb pointed directly at the sun, set to explode at a distance of 100 million miles from it (pretty much at Earth point), taking the entire solar system along with it. The aliens responsible for this bomb were of the opinion that this galaxy wasn't big enough for both of us, and they intended to send us out of town the best way they knew how. However, the facilities for defusing the bomb are available . . . if the Earthlings can figure them out. And the player at his Mission Control terminal is responsible for directing his robot to do just that—the fate of the entire solar system is in his hands! This scenario gave me the overall structure to work from, and the individual complications I would eventually work in would lead to the ultimate solution of the major problem, defusing the bomb.

Complications are the key to a good game, so let's take a hypothetical complication and see how it works. The diamond necklace and the cobra mentioned before will serve as a good example.

Here is the scenario. There are two rooms, east and west. You begin in the west room, where there is a cobra coiled around a diamond necklace. You enter the command TAKE NECKLACE and the computer replies, THE COBRA BITES ME. I DIE ON THE SPOT. SORRY. CARE TO TRY THIS ADVENTURE AGAIN? On the next run-through you try the command KILL COBRA, and the computer replies, TELL ME HOW. At least you're still alive, but you have to come up with some way of killing or otherwise removing the cobra. So you GO EAST, and in the east room you find a collection of musical instruments, including a sitar, a drum, and a flute. You try TAKE SITAR but that doesn't ring any bells, so you LEAVE

SITAR. TAKE DRUM is similarly fruitless, so you LEAVE DRUM. Now enter TAKE FLUTE and then GO WEST. Now try PLAY FLUTE. This time the computer responds, THE COBRA DANCES RHYTHMICALLY TO THE MUSIC AND THEN FALLS ASLEEP. (In a good adventure the commands CHARM COBRA and CHARM SNAKE should work similarly well.) Now you try TAKE NECKLACE and, sure enough, it works. That is a complication, albeit a simple one.

But let's assume that there are actually three treasures in our hypothetical adventure. We now have one, the diamond necklace, but where are the other two? We've already manipulated the drum and sitar and they were not treasures, and there are no other objects around. Here's where you get sneaky. GO EAST and TAKE SITAR again. The computer responds, WHEN YOU TOUCH THE SITAR IT TURNS INTO GOLD. Try this on the drum and it too turns into gold. The trick is, of course, that the necklace is a magic necklace with a built-in Midas touch. And now we have one complete, if simple, phase. You can't get the golden instruments until you get the necklace—everything is interrelated, and in a fairly complicated way (for the sake of explanatory convenience I will ignore the fact that the flute and the snake should also be turned into gold if the necklace were working full tilt, but in an actual adventure game you would have to take those problems into consideration).

Our hypothetical adventure had two complications—charming the cobra and using the necklace for alchemy—interrelated in such a way as to be dependent on each other in a logical progression. That is what I mean when I use the term "phase." A phase is a collection of intertwined complications.

The complications of your adventures should be the aspects that the players get stumped on; the phases should be the aspects that have the player ready to throw the computer out the window in frustration. When a player figures out any given complication he should find that figuring it out is not enough. There should be a kicker in it that further complicates it—compound interest, so to speak. He'll enjoy the game much more if he really has to work for it. Throwing the computer out the window is an accepted facet of adventure playing.

In designing your game you should first lay out the complications, then layer them into phases. You might make it such that until a player successfully completes one phase he cannot progress to the next. An example of that in our snake scenario would be a third room where, for some reason, the player needs a golden sitar to survive. Or you might layer the phases so that successfully accomplishing one is not absolutely essential but makes life easier, perhaps in our snake scenario a third room where you can kill an Indian potentate, but it would have been easier to lull him to sleep with sitar music. And so on. The key is to stay one step ahead of the player. Let the player think he's figured it out, but design the adventure so that there's always one more thing before he really has it down.

Since complications are the essence of an adventure game, you are on your own in coming up with them, but I can give you a few more ideas and examples that can serve as a starting place for your own imagination. One important thing is the concept of the locked door. You shouldn't make it too easy for your player to roam through your universe, and keeping the passageways barred is the best way to inhibit movement. The easiest way to lock a door is literally to lock it and hide the key, but there are other ways of doing it as well. You can cover a passageway with boulders too heavy for the players to lift, then hide a bulldozer somewhere, and maybe even hide the key to the bulldozer. You can make a passageway so small that a player cannot get through it—until he finds a shovel to enlarge the hole. You can have your door so high that the player first has to find a ladder or a rope. In ''Space Derelict!'' I have five different kinds of locked doors: Four are literal doors and one is a ''conceptual'' door (when a player tries to pass through this particular passageway he gets blasted—unless he ''opens'' the imaginary door). There are four methods of opening these varying doors: pressing buttons, using a metal clicker, telepathy, and blasting. Needless to say, the more original your style of locking and unlocking doors, the more fun for the player.

Another complication that's nice to put in is the secret room (which, in a way, is really just another version of the locked door). A secret room can be precisely that; to discover and enter it a player must perform some particular action germane to secret-room finding. You have to give a hint that such a room exists or else the player will never find it, but you needn't make it too easy. In my ''Casablanca'' game the player was made aware of the need of a hiding place through dialog uttered by one of the game characters. To find the hiding place the player had to bribe a beggar. As soon as the money passed hands the player found himself in a secret room behind a brick wall in a corner of the city's bazaar. Any way you want to hide your secret rooms is fine, but make sure you've left a clue somewhere so that the player can get to it eventually.

Here are some other examples of possible complications. In our snake/flute scenario, you might provide the player with a snake, a piece of wood, and a hole-puncher—now he has to figure out how to make a flute and then use it. Or have an alligator guarding a treasure and plant a feather somewhere else—TICKLE ALLIGATOR might be an interesting command (unless your player is foolhardy enough to WRESTLE ALLIGATOR, in which case more power to him). Or perhaps you have vampires to vanquish. Provide the player with a woodpile and a hatchet and let him figure out that he has to MAKE STAKE to create his vampire weapon. Or plant a werewolf in your game, then hide a gun, a bullet-making apparatus, and a silver mine. Make the player work for his victory—don't just hand him a gun loaded with silver bullets. That's what complications are all about.

MAPPING OUT THE AREA

Once you've roughly planned your complications and you have a good idea of the flow of your game, it's time to make a map. How precisely you wish to define your universe at this stage is up to you. You can make measured maps on graph paper, or simply plot out the areas on the nearest scrap sheet. In either case, once your map is finished, *do not throw it away!* No matter how finished you think you are, no matter how many years have passed, no matter how perfectly you think your game is running—somewhere, somehow, there's going to be a bug in it, and without your map as a reference even the most clearly laid-out program will look as though it was written in Chinese after you've been away from it for a while.

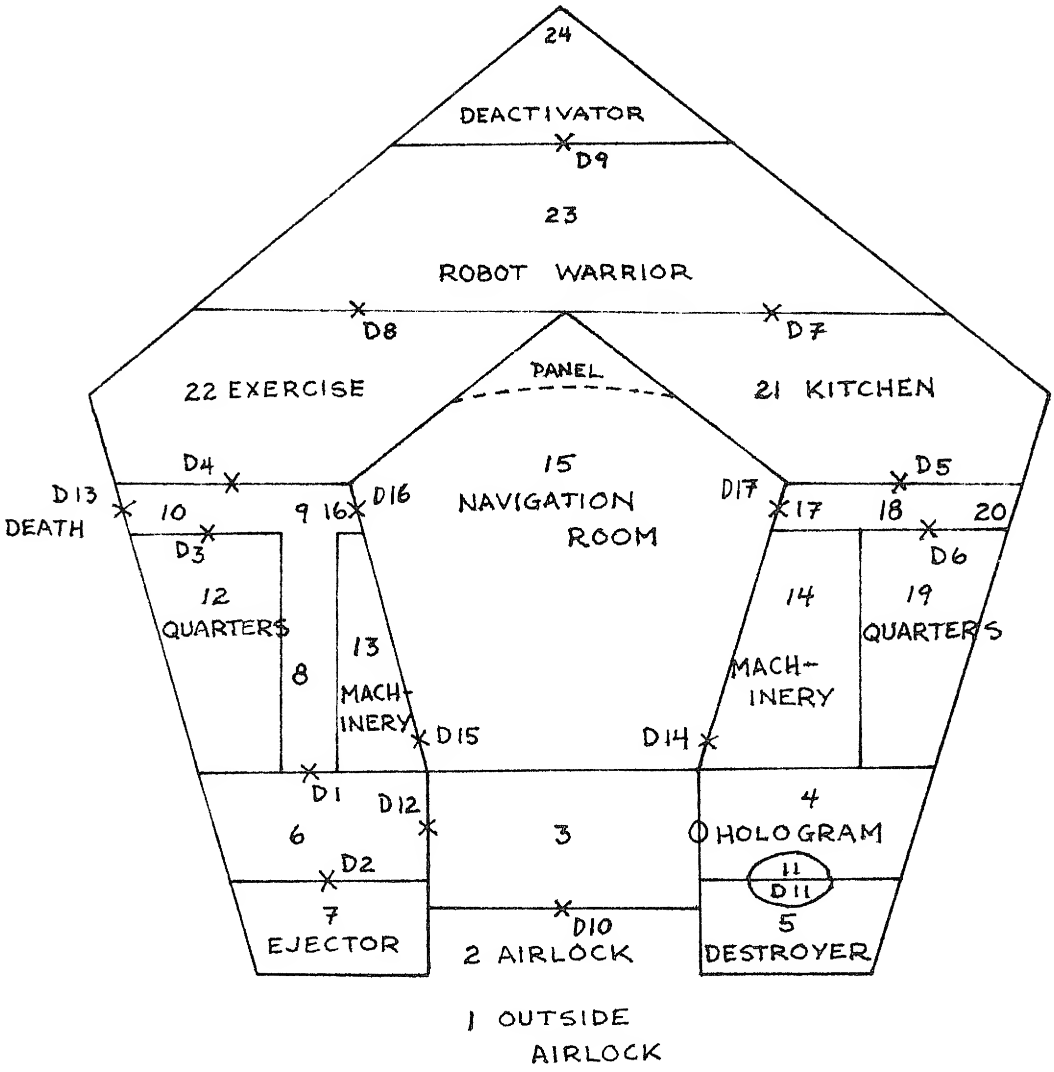
Since every area marked off on your map will appear on the monitor, for the sake of simplicity every one of these areas should be considered a room. A room is any unique area that the player can roam around in, with exits to other unique areas. A room can be an actual room, of course, but it can also be a part of another room, or a street, a cave, a treetop, a boat, or any other area of your universe the player can explore. The reason for this is simple: If you call them all ''rooms'' you can assign them one variable within the program. Call them all something different and you'll get lost in the bookkeeping.

A simple map might consist of three boxes connected horizontally. Room 1 would be on the left, room 2 would be in the middle, and room 3 would be on the right. Let's say that room 1 is a barbershop, room 2 is the backroom of the barbershop, and room 3 is a parking lot behind the barbershop. Let's say further that the objects in room 1 are a barber's chair, a table with some three-year-old magazines, and some shelves with tonsorial equipment. Let's say even further that the barber's chair is a complicated space-traveling device with all sorts of buttons and whatnot—so complex, in fact, that it deserves its own room-type listing on the monitor. In that case, we'd make the chair room 4. To get from room 1 to room 4 the player would say, GO CHAIR. The monitor would change and the player would be facing a whole new area. Again, the reason for this sort of division is simplicity of variables. If you were to keep the very complicated chair as part of room 1, you would find yourself going crazy trying to keep track of it after the GO CHAIR command. This way it's another area entirely, so you can handle it the way you would handle any room. Boiling problems down to their simplest factors is one of the first laws of programming. The fewer things you have to do to obtain results, the better.

So now we have four rooms, two of which are rooms per se, one of which is a parking lot, and the other of which is a barber's chair. Draw a little box in the center of room 1 on your map to represent the chair, and now label the four rooms 1, 2, 3, and 4. If you've got enough space (and I certainly recommend this) you can also label the rooms descriptively—Barbershop, Backroom, Parking Lot, Barber's Chair. Now you've got a very clear hypothetical map of four areas with nothing in them. But if this were a real adventure you had designed, you would have numerous items with which to stock the rooms. It's best to make a list along with the map, and on that list start with room 1; identify it, then list the things that will be found there. A few descriptive lines about the room might also be helpful when it comes time to program them; it helps set the rooms in your own mind.

Now you're ready for your first real programming thingamabob, a variable called R. In our adventure programs the variable R will refer to the room number. In our example there are four possible values for R. If R=1, then the room is the barbershop, room 1. If R=4 then the room is the barber's chair, room 4. If R=2392, you haven't been following closely enough, or else you've added a few extra rooms to our haircut scenario.

The map of ''Space Derelict!'' is reproduced below. There are 24 rooms in it, so in this game R will range between 1 and 24. Some of these rooms are rooms, others are hallways, one is a door, others are parts of hallways,



24
DEACTIVATOR
D9
23
ROBOT WARRIOR
D8
D7
PANEL
22 EXERCISE
21 KITCHEN
15
NAVIGATION ROOM
D4
D5
D13
DEATH
10
9
16
D16
D17
17
18
20
D3
D6
12
QUARTERS
8
13
MACHINERY
14
MACHINERY
19
QUARTERS
D15
D14
D1
D12
6
3
4
HOLOGRAM
11
D11
D2
7
EJECTOR
D10
2 AIRLOCK
5
DESTROYER
1 OUTSIDE AIRLOCK

and one (which happens to be numbered 1) is just the starting point of the game, located in space outside the space ship.

You will see on this map various written indications of which room is which, and a few other things we'll talk about later. You'll also see a series of numbers ranging from D1 to D17. These numbers refer to doors, 17 in all. In a number of cases there are no doors between rooms, but in more cases than not there are. As the player goes through the game he will have to open all these doors (we'll discuss how later). The important thing for now is that we have just come upon our second variable, D, and this one I'm afraid is quite a bit more complicated than our friend R. So we'll have to break to a new section, aptly entitled:

SIMPLE ARRAYS

Arrays are not particularly complicated, but you'd never know it by reading most explanations of them. The best way to get the hang of them (and this applies to just about every facet of programming) is to use them.

Arrays are a convenient way of organizing our variables so that we can perform various interesting operations on them. An array is a variable that looks like this: D(14). What we're doing here is using one variable name, D, but giving it a lot of different possibilities. In fact, in "Space Derelict!" where D refers to doors, we are going to give it 17 possibilities, so our D variable can range from D(1) to D(17). D(1) is door 1, D(2) is door 2, etc. We could, of course, use D1, D2, . . . D17, but we'd be cheating ourselves out of some neat possibilities. Imagine that we wanted to find out the values of D(1) through D(17). We could say

```
FOR X = 1 TO 17: PRINT D(X): NEXT
```

Without arrays we'd need seventeen different PRINT commands.

For the computer to allot the appropriate memory space for an array, you must first tell the computer how big your array will be. You do this by declaring the dimensions of the array. In our case now the proper dimension is 17, so we tell the computer

```
DIM D(17)
```

and now we can have 17 different doors to do with as we please.

Arrays can be a lot more complicated than that if we wish, and we'll be using more complicated ones later in our poker game—e.g., V(3,4), where the array has two parts. An array can also look like this: NW(1,0,3,4). There is a limit to the number of dimensions in an array, just as there is a limit to the number of variables your machine will hold, based on the finite number of bytes you paid for when you bought your machine. Without going into too much detail here about memory-eating variables (you can experiment with this on your own time), suffice it to say that although you don't have too much to worry about, it's something to keep in mind when you're doing your dimensioning.

The other important thing about dimensions is that almost everything the computer does begins with the value 0. So in our DIM D(17) command we have actually set aside 18 values for D, from 0 to 17. Setting it at DIM D(16) would have given us 17 values, from 0 to 16 inclusive, but in a case like this it's simpler to use tangible numbers, since we're talking about tangible doors—and I find it hard to master thinking of 0 as the first number of a series. It takes time and space for the computer to sort out variables, so using the precise numbers including 0 can be helpful when processing speed is an issue. For example, there would be no point in our saying DIM D(30) when we're using only 17 Ds to get the job done; the computer is going to set aside those extra places even if we don't use them, so you might as well keep things as tight as possible. But ignoring that 0 factor usually won't hurt, providing there aren't too many variables to keep track of. However, if you really want to be tidy you can use the command OPTION BASE 1, which would start all the arrays at D(1) rather than D(0). The OPTION BASE command applies to all the arrays in the program and must be set before you otherwise use any array variable.

In "Space Derelict!" any given door is going to be either open or closed, so any given D(X) is either going to equal 0, closed, or 1, open. If D(4) = 0, then door 4 is closed. If D(4) = 1, then door 4 is open. So now we have 17 doors, each of which is either open or closed; i.e., we have DIM D(17), where D(X) = 0 or D(X) = 1.

That wasn't so bad, was it?

PLANTING THE OBJECTS TO BE MANIPULATED—STRING VARIABLES

So far we've mapped out the rooms and marked the doors. The only major things missing from the physical structure of our universe are the movable objects lying around that the player can play with.

When I did my first adventure I gave the movable objects the variable name of CO, as in COnstant, since those objects would remain invariable throughout the game, and I've stuck with it. Actually we're talking about CO(X) because we have a number of COs lying around, and CO(X) refers to any particular one of them. In "Space Derelict!" there are 16 movable objects, or COs, so let's set it in our minds now:

```
DIM CO(16)
```

CO(1) is object 1, CO(16) is object 16, and the other numbers correspond to all the objects in between.

Unfortunately, as you can see, we now have numbers for all the objects but no names for them, so we have to flip over to the other side of the coin and become acquainted with string variables, in our particular case the string variable CO$(X).

String variables work much like numeric variables; R and D(X) were numeric variables—they were equal to numbers, R=4 or D(5)=0. The difference is that a string variable can contain anything you want it to contain. A string variable is the contents of a set of quotation marks. If we were to type into the computer the line

```
F$="ABRAHAM"
```

then ABRAHAM would be the contents of string F$. The dollar sign tells the computer that F$ is a string. If we were to type in

```
G$="2001"
```

then 2001 would be the contents of the string G$. If we now asked the computer to

PRINT F$
the computer would respond with
ABRAHAM
If we typed in
PRINT G$
we would get the result
2001
In this situation the computer doesn't care what's in the quotation marks and will simply hold those contents, be they numbers or letters or a mixture of both, and read them back to you whenever you want them.

String variables can be manipulated in a number of ways, most of which we'll discuss later. At this point it is enough to remember that a string variable looks like any variable with a dollar sign after it (e.g., F$) and that string variables conform to all the regular variable rules. We can dimension an array of string variables just as we dimension a numeric variable. In "Space Derelict!" we have 16 movable objects, each one of which has a name, and we are going to assign the string variable CO$(X) to each one. Once again we set a dimension for our array of objects, this time
DIM CO$(16)
In any adventure, the numeric CO(X) variable will correspond to the string CO$(X). Now I'll put them together to show you what I mean.

In "Space Derelict!" CO$(1)="Odd piece of metal. " After I've set this into my program, if I were to type
PRINT CO$(1)
the computer would respond with
Odd piece of metal.
However, in my game I want to know more than just *what* a particular object is; I also want to know *where* it is. That's where my corresponding CO(X) comes in. The value of CO(X) will tell me the location of CO$(X).

In "Space Derelict!" there will be two possible locations for any CO(X): Either an object is lying about in any one of the 24 rooms, or my robot is carrying the object. If my robot is carrying it, then I will assign it a value of 1. Let's assume my robot is carrying the odd piece of metal. In that case
CO(1)=1
Regardless of which CO(X) movable object I am talking about, if that CO(X)=1, then the robot is carrying that object. Since we've just established that CO(1)=1, we know that the robot is carrying the odd piece of metal. Note that this logic works in both directions: If we know that the robot is carrying the odd piece of metal, then we know that CO(1)=1.

The other possible location for a movable object is lying around in any one of the 24 rooms. To make this as simple as possible we will set 100+R as this value for CO(X) (and believe me, that is simple—don't throw the book out the window yet). We know that R equals a room number; if R=19, we know we're referring to room 19. So if our odd piece of metal were lying around in room 19, then CO(1) would equal 119 (100+R, or 100+19). You'll see the beauty of this when we get into the program and start picking things up and dropping them off. Let's set this now:
CO(1)=119
and play with it a minute. Our robot is in room 11, let's say, which means that R=11. Our player enters the command TAKE ODD PIECE OF

METAL. At this stage in our program the computer will know that the odd piece of metal is CO(1), so the computer will try to figure out if CO(1)=100+R. Since CO(1)=119 and we're in room 11, therefore R=11 and 100+R=111; so the computer will be able to respond conveniently THAT IS NOT HERE AT THIS TIME.

Or let's imagine that we are in room 19 and CO(1)=119. In this case R=19. This time if we try to take it, the computer reads the 100+R, which indeed does equal 119, the value of CO(1). The program now will change the value of CO(1) to 1—remember, if we're carrying a particular CO(X), then that CO(X)=1. How this is all done in the program you'll see later, but the fact that it can be done with these variables is the important thing now. If you're thoroughly confused, hang on a minute, because it might make more sense when you see it laid out in front of you.

THE "SPACE DERELICT!" SHOPPING LIST—CO(X) AND CO$(X) IN ACTION

The problem with trying to explain complicated ideas like the preceding ones is that if we present examples first, they don't make sense without a previous explanation, but on the other hand if we start with the explanations they don't make much sense until you see them in action. Computers are complicated animals and even the simplest aspects, when described in writing, are going to look like gobbledygook until you let them settle into your head for a while. There are always a million things going on at once, and it takes some time to get them all straight.

In this book there are going to be two kinds of program listings. The first kind is hypothetical and will look like this:

```
100 GOTO JAIL:REM GO DIRECTLY TO JAIL
110 IF X=$200 THEN DO NOT PASS GO
```

This kind of listing will be used to explain various programming concepts relative to nothing in particular except those concepts.

The second kind of listing will look like this, the name of the program flanked by asterisks:

Space Derelict!

```
2260  CO$(1) = "Odd piece of metal. "
    : CO(1) = 119
2270  CO$(2) = "Small lead pentagon."
    : CO(2) = 104
2280  CO$(3) = " Crystal pentagon.  "
    : CO(3) = 107
2290  CO$(4) = "Unidentified weapon."
    : CO(4) = 121
2300  CO$(5) = "   Pentagonal box.  "
    : CO(5) = 105
2310  CO$(6) = "  Crystal star.     "
    : CO(6) = 0
2320  CO$(7) = "Small laser pistol. "
    : CO(7) = 1
```

```
2330  CO$(8) = "Small flashlight.   "
    : CO(8) = 1
2340  CO$(9) = "Large blade knife.  "
    : CO(9) = 121
2350  CO$(10) = "  Metal bucket.     "
    : CO(10) = 121
2360  CO$(11) = "Ruined alien robot. "
    : CO(11) = 105
2370  CO$(12) = "Strong hemp rope.   "
    : CO(12) = 0
2380  CO$(13) = "Giant five-legged robot."
    : CO(13) = 123
2390  CO$(14) = "Small area of water at edg
        e of ice.     "
    : CO(14) = 0
2400  CO$(15) = " Filled bucket.     "
    : CO(15) = 0
2410  CO$(16) = "Deactivated robot."
    : CO(16) = 0
```

The difference will be in those asterisks in the heading, plus the distinctive typeface. If a listing has a heading, it will be either *Space Derelict* or *Five-Card Draw* and the listing will be part of one of these two programs. If you want to enter these programs into your machine, then type in those listings whenever they appear. There will be no uninterrupted listing of either program in the book, so to get them up you will have to enter them a piece at a time.

If you're interested in getting "Space Derelict!" up and running, you'd better get thee to a computery and start typing now! Copy the listings above for 2260–2410 EXACTLY as they appear (and pay particular attention to the spacings used within the quote marks). These lines in the "Space Derelict!" program establish the name and position of each movable object (the use of the colon allows us to put these two inextricably related variables on the same lines, side by side). This piece of program will be executed before the actual game play begins, in order to set our two variables CO(X) and CO$(X). Let's see what we have here.

Line 2260 is our odd piece of metal, and as you can see, it really does have a CO(1) value of 119. That means that the odd piece of metal is in room 19 (again, that's 100+R, where R is equal to the number of the room, in this case 19). Lines 2270 through 2300 follow a similar pattern. The small lead pentagon that is CO$(2) is located in room 4, obviously, because CO(2)=104, i.e., 100+R or 100+4. And so on. We'll skip 2310 for a minute and look at 2320. Since CO(7)=1, we know that our small laser pistol, CO$(7), is being carried by our robot. Our small flashlight is similarly in the hands of our puppet robot, since CO(8)=1. The mean-sounding giant five-legged robot of CO(13) is, of course, lying in wait in room 23. That leaves those five lines (2310 and 2370–2410, excluding 2380) where CO(X)=0. Earlier I said that there were only two possible locations for any CO(X), either in the hands of our robot or in a room, a CO(X) value of either 1 or 100+R, respectively. So shoot me, I lied a bit.

The point here is not to be misleading but to show you that in an adven-

ture you can put an object anywhere you want simply by assigning any meaningful number to it. Any 100+R value places the object in a room, and a value of 1 places it on your puppet's person, but that leaves an awful lot of numbers for you to play around with if you are so inclined. Perhaps you have a few roving monsters in your adventure, and these monsters are capable of picking up objects. Let's say there are 10 monsters. If any of them were to be in possession of a CO$(X), then you could set it in your program as:
CO(X)=200+M:REM 200 INDICATES CO$(X) IN HANDS OF MONSTER, M INDICATES WHICH ONE
Or say you want to add locations to your adventure more specific than the rooms. You might have room 14 in which there is a grand piano, and you want to be able to hide a celadon croissant chopper, CO$(22), in that piano. You could assign, say, a value of 2 to the piano, and therefore CO(22)=2 would put that strange little piece of kitchen equipment right where you want it. The thing to remember is that the values of CO(X) can be flexible. I've assigned the number 1 to items of inventory (i.e., items on the person of my puppet robot), and 100+R to objects in rooms. You can add as many variations to this as you wish.

All of which doesn't explain my use of CO(X)=0. The value 0 is very specific to these individual items; in essence these items do not exist in the game until certain other criteria are fulfilled. The crystal star of 2310 is well hidden, as is the rope of 2370. When the player finds them, their values will be changed to the appropriate 100+R. CO$(14) is water that will exist only if the player is clever enough to melt some ice, and the filled bucket of 2400 will at some point replace the metal bucket of 2350, as soon as the player thinks to fill CO$(10) with the water he has created at CO$(14). These 0-value CO(X)'s are reserves lurking around, to be called up to action when the time is right. Don't forget that concept of complications. If we were to leave everything in plain sight it would be just too easy; in "Space Derelict!" my 0 value of CO(X) is my catchall hiding place for some of my more devious complications.

Now, having entered lines 2260–2410 into your computer, you can save these lines as "INTRO" and turn the machine off. You now have a complete list of the manipulable objects in the game—the "shopping list," if you will—and you also have your first listings on your way to assembling the entire game. You are on your way and loaded for bear.

2

LAYING OUT THE GAME— THE DESIGN FORMAT

"SPACE DERELICT!"—THE SCENARIO

You know a little bit about "Space Derelict!" already; now I'll fill in the blanks and outline the play of the game. The level here is pretty basic, with not that many rooms or manipulable items. It was designed primarily as an instructional tool, and as such I've tried to keep the complications manageable. But the nature of an adventure remains the same regardless of its complexity. In fact, "Space Derelict!" would make as good an introductory game for players as it does for game designers. So let's run through it and see how it goes.

After an introduction explaining how the game is played and setting up the player's role as Mission Controller, our puppet robot arrives at the UFO (room 1 on the map on page 14). There is nothing to be seen but an airlock ahead of him. When he enters the airlock (room 2) he is faced with a locked door and a panel of three different-colored buttons. If he presses the right button, the door opens; if he presses the wrong button, he either shuts a door behind him or else is blasted into oblivion. Assuming he has pressed the right button, he now enters the ship (room 3).

Once inside the ship our robot encounters a western door that cannot be opened. He has no alternative but to travel east (to room 4), where he will find a small lead pentagon, a hologram, and a strange device. The hologram proves unintelligible, but as he approaches the strange device a message is broadcast from the inhabitants of the planet Krenn. Our friends the Krennians now tell us that the ship is a bomb that will blow up the solar system; it can be defused, but it is doubtful that we'll be able to do it successfully. End transmission. Needless to say, our robot picks up the small lead pentagon; this turns out to be a clicking device that will open some of the doors.

Room 11 to the south is a passageway into room 5. If the robot looks into room 5 from room 11 he can see the debris of a destroyed robot. If he enters room 5 he is blasted into oblivion. At this point there is no way he can enter that room and survive to brag about it.

Our robot now heads back to room 3. By clicking the pentagon he opens the previously locked western door and gains access to empty room 6. Another click opens the southern door (but not the northern one). Entering the southern room he finds a crystal pentagon and another panel of buttons. These buttons serve no earthly purpose but to cause mischief; pressing them invites another blast into oblivion.

This crystal pentagon is an interesting little gizmo. Our robot recognizes it as organic, and eventually our player must discover that it is a telepathic door opener. If the robot thinks (or says) ''OPEN'', then a door will open. A sneaky trick—too sneaky?—and it will take the player a while to figure it out. This telepathic door opener also allows entrance into room 5 back in the other direction, where the player will find a pentagonal box, quite empty. We are now ready to progress into the bowels of the ship.

The hallway (room 8) leads to the crossroads (room 9). The robot can follow a set of footprints that lead to a western door (and another oblivion blast). The room south of corridor 10 is the crew's quarters, now empty. The door at area 16 opens into the control center of the ship (room 15). Here our robot finds a locked panel of controls and a crystal star. Unfortunately, the crystal star won't budge when he tries to remove it.

Two southern doors open into machinery rooms (13 and 14), neither of which yields any information. Exiting northeast brings our robot to corridor 17–18–20. Going south from 18 is another crew's quarters, where our robot can pick up the odd piece of metal we talked about in the previous chapter. North of 18 is the kitchen, where our robot can obtain a metal bucket and a knife.

If our robot goes north from here into room 23 it's curtains, so let's backtrack through the main control room first. Using the odd piece of metal, our robot can now remove the crystal star. Down the corridor and up to room 22 is some sort of exercise area with climbing ropes hanging down, a swimming pool totally frozen (it's cold in outer space), and even a marked-off court for some kind of ball game. Climbing the ropes is fruitless, but if our robot is so directed he can cut the rope and use it for other purposes (all of which will prove futile—the rope is a red herring). Melting some of the ice and putting it in the bucket will be a good idea, however. Now if we go north to room 23 we find that a robot is waiting to dish out the traditional oblivion blast. This robot is almost omnipotent. As it turns out, the only thing that will put him out of commission is a bucket of water over the head. We can now proceed up to room 24.

Unfortunately, neither our clicker nor our telepathic device opens this final door, but a good blast from the laser pistol does the trick. Once inside we find a panel with a slot in it. This is the bomb defuser. First we must insert the crystal star into the pentagonal box (clues have been given as to the advisability of this). Then we insert the filled box into the slot. And a message from the Krenn congratulates us on our cleverness but warns us that we will hear from them again. End of game.

It doesn't look all that complicated laid out like this (as I said, this game was specifically written as an instructional tool), but take my word for it—nobody's going to walk through this baby in five minutes. The biggest pain in the patoot will be the offhanded blasts into oblivion; the player would be well advised to save the game whenever he's trying any new maneuvers. Of course, now that you've read the description *you* could walk through it in one sitting, but your situation is different: You're learning how to create other adventure games. Let's go on now to the actual program format, 100 percent guaranteed for any adventure you set your mind to. A simpler setup you couldn't imagine.

THE ADVENTURE MODULE EXPLAINED

When you get right down to it, there isn't very much going on in an adventure at any one time. The player is in a room; there may or may not be other objects in the room with him. And the player may or may not be carrying certain items. That's all there is to it. Most of adventure game module is concerned with one aspect or another of this basic situation and boils down mostly to moving the player and/or objects from one room to another.

The format of the basic module breaks down like this:

Introduction/housekeeping routines
Room layout routine
Room objects subroutine
Inventory subroutine
Game-saving routines
Miscellaneous-business routines
Winning and losing scenarios
Conversation with the computer

We'll take these one at a time and talk about what they do. Keep in mind that they are valid for any adventure, not just "Space Derelict!"

Introduction/housekeeping routines. The introduction is just that. It includes such things as the game title and author and if necessary some instructions on how to play the game. Further, you might need to set up certain plot elements to give the player some background. This is where all that would take place.

By housekeeping I mean setting variables (also known as initialization). The housekeeping routine would include, for instance, the list of CO(X) and CO$(X) values we talked about earlier plus a bunch of other variables we'll talk about later. This subroutine in "Space Derelict!" takes up only 50 lines, and even an adventure five times more difficult wouldn't require that much more housekeeping space. It's this sort of simplicity that allows the game designer to concentrate on the play of the game rather than data crunching.

Room layout. The room layouts are the physical descriptions of the rooms that will appear on the video monitor. Because the adding of the room layouts to the main program might eat up more memory than we have available, we store this information on the disk in separate disk files rather than in the program. We'll go into the problem of limited memory and how to get around it shortly. A good working knowledge of disk files can come in very handy in much of your programming, games or otherwise.

Room objects subroutine. An offshoot of the room layouts, this short piece of business reads the CO(X) values of all the movable objects and decides whether each object belongs in the room at hand.

Inventory subroutine. Another short piece of business much like the room objects subroutine. This time the CO(X) values are read to determine whether a particular item is on the person of the puppet robot—i.e., whether the various CO(X) values equal 1.

Game-saving routines. These are exactly that. One of these routines allows you to save a game in progress; the other allows you to pick up a game where you left off. In "Space Derelict!" these routines total about 15 lines each; their length depends on the number of variables in a given adventure. Again, it's hard to imagine a reasonable adventure with very many more variables than we have here.

Miscellaneous-business routines. The less miscellaneous business in your adventure the better. It's best if you can contain the game within the confines of the major routines. But sometimes this just isn't possible and a miscellaneous area in your program becomes a necessity. In "Space Derelict!" I use this area to monitor the attacks of the five-legged robot in room 23. The problem with miscellaneous areas is that it can be hard to get them to work right, which is why they are best avoided.

Winning and losing scenarios. You need some place to go in your program if the player wins. Also, if you've built in any setups where a player could get killed, you need a resolution for that, too. That's what these are all about.

Conversation with the computer. Roughly 500 lines of "Space Derelict!" are devoted to the player's keyboard conversation with the computer, and any expansion of this number would result in a proportional improvement in the adventure. The more conversation the computer is capable of, the better the game. In this adventure the player enters either one- or two-word commands, and the computer responds accordingly. If the program recognizes the command, an appropriate action is taken, and the game progresses. If the program only partially recognizes the command, the player is advised that he's warm but not quite on the money. If the command is a complete bust, the player is told to try something else. In "Space Derelict!" the program recognizes about 50 verbs and well over 100 nouns; more importantly, it knows what to do when it hears them.

MEMORIES ARE MADE OF THIS

Imagine you are sitting at the keyboard of a new 64K Model 4 that you've just taken out of the box and plugged in. You take a minimal look at the directions and figure out how to turn the machine on by booting TRSDOS. Then you load BASIC and you're ready to go. Now you might be tempted to think that you have 64K of memory to fool around with, but there's a Basic command called MEM that will in fact tell you down to the last byte how much memory is available to you in the machine. With this brand-new 64K Model 4, with TRSDOS and BASIC booted, if you type

```
PRINT MEM
```

you get the response

```
29424
```

and it's Whoa, Nellie! Who left the barn door open? There're some 35000-odd bytes missing somewhere. You might be tempted to shake the machine a little bit thinking these bytes might have gotten loose inside somehow, but the fact is that you really do have only about 30K available to you. (Aren't you glad you didn't buy the 16K cassette version? And don't you wish you bought the 128K super version?) The reason for this is that both TRSDOS and Basic (and a few other odds and ends besides) have to go somewhere, and that somewhere is in that 64K. But don't despair, because what you have available to you is still more than enough to do just about anything you want it to do. You just have to learn a few tricks of the trade to make the most of it.

If we were to attempt to write "Space Derelict!" as one whole program on the Model 4 we would find that sooner or later we would get a message from the machine that we had run out of memory (I know this, because I tried it). Part of the reason for this is the deliberately oversimplified structure of the game to make it understandable, which necessitated making lines clearer and hence longer than they might otherwise have been. But even trimming the game down to the bone using the structuring techniques we'll be discussing in Part 2 of this book wouldn't have made that much of a difference. We might have gotten it to fit, but that's no guarantee that the next adventure or the next after that would also fit, so a solution to this memory problem outside of mere line trimming becomes a necessity. In finding this solution we have only one requisite—that whatever it is, it doesn't hurt the play of the game. Fortunately, the solution is a simple one, roughly about five inches square with a hole in the middle. Maybe our machine memory can't hold more than 30K of our programs, but our disks can hold four or five times that amount. If we can break our program down into manageable bite-size pieces and store them on the disk rather than in memory without affecting the play of the game, we've got it made. We can, and we do.

We are going to use the disk for our solution in two ways. The first and easier one is the division of the whole monster program into four smaller ones. The first of these smaller programs will contain our introduction and housekeeping information. The second one will contain our winning scenario, while the third will contain our losing scenario. The fourth and largest of these programs will contain everything else, including the major segment of the conversation with the computer. And to connect these four programs we will use the command called CHAIN, which automatically links programs in memory. The program we're running is removed from memory when the CHAINed program is loaded in, so essentially our only memory limitation now is the storage size of the disk.

Whenever we CHAIN programs we have the option of carrying variables from one program to the next or simply starting afresh with the next program. Additionally, we have the option of starting the new program at the beginning or anywhere we want in the middle. In fact, we even have the option of CHAINING two programs by MERGEing them, overlaying lines of the second program on to the one already in memory. In our adventure we will be doing all those things with the exception of MERGE.

A CHAIN command looks like this:

```
1000 CHAIN "GANG", ALL
```

If we were running a program and came to this line, the machine would load in the program called GANG off the disk, carrying ALL the variables values from the present program and begin running GANG at the very first line, erasing whatever program we had been running from memory but nonetheless holding the values of that program's variables. If we had typed in the line like this

```
1000 CHAIN "GANG", 330, ALL
```

the machine would still load in GANG carrying all the variables and erasing the program already in memory, but programming would commence in the GANG program at line 330 rather than at the beginning.

Using CHAIN is pretty simple, but it does have one or two minor drawbacks. First of all, it can be used only on programs SAVEd in the ASCII format. Now, if you were so inclined you could read your owner's manual from now to kingdom come and never truly figure out what this ASCII format is all about versus whatever the other format is available for saving programs. Without going into too much detail, saving a program in the ASCII format means that the program is placed on the disk so it is understandable in more ways than one. In the computer world, ASCII is the coin of the realm. The initials stand for American Standard Code for Information Exchange. ASCII codes are roughly standard from one brand of machine to the other, but they're about the only thing that is, and even that doesn't allow two dissimilar machines to communicate with each other. Nonetheless, using ASCII allows programmers familiar with one sort of machine to have at least a starting place when they try to figure out another machine. ASCII also helps nonconversant computers to communicate over telephone wires. The alternative to saving programs in the ASCII format would be to use the special compressed program format that the TRS uses in its own universe (in fact, regardless of whether or not a Basic file is stored on disk in ASCII, it is loaded into memory in the special TRS format). The major difference in the compressed format is that the TRS substitutes numeric abbreviations called "tokens" for the various Basic commands and functions. But CHAIN can only recognize ASCII programs on the disk, so ASCII SAVEs become a must for us.

Saving a program in the ASCII format is no big deal. All you do is add a comma and the letter A after your save command. So in "Space Derelict!", with our four small CHAINed programs, we'll always save them thusly:

```
SAVE "INTRO", A
```

thus assuring ourselves of having the right sort of program file on hand when we need it. One of the trade-offs in saving "A" files is that they take up a bit more space on the disk, but that's a small price to pay for programs that can talk to each other.

Another problem with the CHAIN command is that when we want to pass variables we need another command as well, called COMMON. If we want to CHAIN program number 1 with program number 2, and number 1 has the variable X, the value of which we have to pass to number 2, we have to put in the statement

```
10 COMMON X
```

in both programs. Actually, COMMON is optional in some cases, a requirement in others, but it's only one line, so if you are CHAINing, it is a good practice always to include it, just to be on the safe side. With more than one variable you would COMMON them all, as in

```
10 COMMON X, Y, Z$, D(), FRED$()
```

The use of the empty parentheses allows us to carry the values of any arrays, so this line would carry D(0) through D(whatever), without us having to specify what that whatever is. Because "Space Derelict!" uses arrays, our COMMON lines will look very much like this one, as you'll see shortly.

I said earlier that there were two ways we used the disk to solve our memory problem. CHAIN was one of them; sequential-access disk files are the other.

Disk files are very much like literal manila files in a filing cabinet: they are repositories for data in an ordered, accessible format. Disk files are used for such things as lists of addresses where the information is repetitive and in a precise format; each disk file might contain one person's data, giving that person's name, address, telephone number, and occupation, always in the same order. Or each file could contain the data for 20 people in that same precise, rigid order. When it came around to sorting these files you would always know where to look for whatever you wanted because it would always be in the same place: The name would always come first, then the address, etc. We're going to use this same concept in adventures.

The structure of disk files is not complicated, but it is extremely rigid. If precise handling is not accorded them, they will come back and slap you down with the most amazing sort of unexpected results. Unless you're already fully versed and capable in the file department I would advise hitting the manuals one more time until you do feel comfortable with them. I'll do my best to explain them for my own purposes, but they are much broader in scope than what we'll be doing here, and they can solve an awful lot of programming problems for you if you know how to work them imaginatively and accurately.

In our adventure we are going to use sequential-access disk files, as compared to direct-access files. A sequential file is always read from start to finish, whereas direct-access files are open to more idiosyncratic approaches (and that is all we will be saying about direct-access files in this book). A sequential file is always read from the first *field*—which is what we call a piece of information in a file—to the last field. And when we manipulate these files the computer in a sense really does read them à la manila folders.

The process is straightforward. The computer opens a file, reads it, and then closes it up and puts it away again. So let's say that a sequential file contains the following bits of information:

Mother of pearl.
Emperor of Russia.
Yellow ribbons.
1
0
0
1

When we ask the computer to open up that figurative manila file it reads these pieces of data in exactly this order, and we can now do whatever we want with them. At the moment they appear to be nothing more than a series of seven unrelated words and numbers. It is up to the program to make some sense out of them.

Creating a disk file is no mean feat, requiring a number of complicated and perhaps unfamiliar commands. We are going to use disk files to do two things for us: store all the information on the rooms in our adventure, and save games in the middle of play. In this chapter we'll concentrate on the more complicated creation of the room files.

PROCEDURE—THE PROGRAMMING BEGINS

You now have designed your adventure in all its complications, and you're ready to put it on a disk to see how it runs. For almost any program other than an adventure it would be imperative that you have the programming all done on paper, line by line, before trying to load it into the computer. Sitting at the keyboard wasting electricity is no time to be figuring out the lines of a program. But adventure programs are different. Once you have one finished program in hand, you can use that as your outline for the next one. Only the names are changed to protect the innocent. With a good, complete design in mind, you can actually do most of your programming at the keyboard. Not that it will come out perfect the first time—you can't expect that kind of success with even a one-line program!—but with the techniques I'll describe later you should get it running like a charm by the third run-through, not a bad debugging experience. You might set aside about three months of evenings for the entire process.

The first step in programming, after setting the movable objects (as we did in Chapter 1), is laying out the rooms. You could conceivably start out somewhere else in the program, but I wouldn't advise it. When a player is going through an adventure he spends the majority of his time simply poking through the various rooms, looking at things and examining them, going back and forth trying to get the feel of the game universe. The computer conversation segment of the module must be as comprehensive as possible in predicting exactly what the player will ask about. Doing the room layouts first allows you to have in hand an exact copy of what the player will see in his travels, allowing you to handle the conversation segment accordingly. The better you handle this at the start the less time you'll have to spend debugging it later.

As I said earlier, we're going to use sequential-access disk files to store our room descriptions. You'll recall the room variable R, and how R could have any value from 1 to 24, telling us what room we were in of the 24 rooms. Now you'll begin to see exactly how this works as we create 24 separate sequential-access files, one for each room. This may sound like a monumental undertaking, but I assure you there's nothing to it once you get going. In fact, all it takes is the very short program that follows, which I call "RMAKER." "RMAKER" can be used for any adventure you might want to try, easily adapted if need be, so it's a useful little utility to have in your bag of tricks.

*"**Space Derelict!**"*

```
10 OPEN"O",1,"R1:1"
20  FOR X = 1 TO 3
30 LINE INPUT R$(X)
40 WRITE#1,R$(X)
50  NEXT
60  INPUT N
70 WRITE#1,N
80  INPUT S
90 WRITE#1,S
100  INPUT E
110 WRITE#1,E
120  INPUT W
130 WRITE#1,W
140 CLOSE1
```

What this program does is take input from the keyboard and make a disk file out of it. Line 10 OPENs a file. The "O" stands for Output and tells the computer that we want to put data into a file rather than take data out of one. The "1" after the "O" tells the computer to use memory buffer number 1. A buffer is actually a small area of memory put aside just for disk files (some of our missing 64K!). If we have only one file working at a time, which is all we'll ever have in adventures, we'll always want buffer number 1. We don't have to worry about the whys and wherefores of multiple buffers because we won't be using them, but they are clearly explained in the manuals if you're interested. Finally, the "R1:1" in line 10 gives our disk file the name R1 and tells the computer to store it on the disk in drive number 1, which will be the disk that will contain all the parts of our "Space Derelict!" program.

We now have an open file called "R1" waiting for us to put something into it, and lines 20–50 set up a loop for us to begin doing just that. What we're going to put into that file is, first, up to three lines of text description of each room, followed by markers to tell us whether we can go north, south, east, or west. For "Space Derelict!" three lines of text (each program line can have up to 240 characters) is more than enough to adequate descriptions, but if you're feeling really chatty you could increase the number of lines as much as you want. Line 20 begins our FOR . . . NEXT loop that will run us through lines 30 and 40 three times. Line 30 actually gets the typed INPUT from the keyboard, storing it as R$(X). We don't bother to prompt ourselves with a line like "Enter text" or somesuch because there's so little going on here we can absorb it very quickly. The INPUT command stops program execution until we actually put something into the computer and then press Enter. In this case whatever it was that we Entered will become R$(X). You'll notice, however, that we're using something called LINE INPUT rather than plain old INPUT. Putting that LINE in there allows us to save such things as commas and quotation marks in our INPUT strings, which would have been impossible otherwise—try it and see what happens. The quote marks are no problem in the text for "Space Derelict!" because we don't use them, and in fact plain old INPUT would accept a single quote mark ('), which would do perfectly well in most cases anyhow. But try to write a coherent room description without using commas! The problem with commas in a regular INPUT is that the computer sees a comma the same way it sees an Enter, as a carriage return, which in some cases can be very useful but which wreaks havoc on what we're doing here. Fortunately, LINE INPUT solves the problem, so we needn't worry about it.

In line 40 the R$(X) we've just LINE INPUT is put into our new "R1" disk file (or, more accurately, into the memory buffer preparatory to going into the actual disk) via the WRITE command. "WRITE# 1" followed by a comma and a variable tells the computer to put whatever is contained in that variable into the file in buffer number 1 (or any other specified buffer, if you're starting to get fancy on me). So between lines 30 and 40 we can INPUT up to three lines of 240 characters each of room description, for a total of 720 characters. This means a maximum of about nine lines of text, and again, if you need more, just amplify the X in line 20 to whatever you want. If you need less than the three R$(X)s, all you have to do is press Enter without any INPUT, creating that particular R$ as what is called a null string—that is, a string with nothing in it (R$(X)=""). And that's half the battle.

The other half is the INPUTing of the directional data. In each room we either can or cannot go in a given direction. When we create our room description files we include that information, and we do it quite easily. If we can go in such and such a direction, we give that direction a value of 1. If we can't go in that direction, we merely press Enter, thereby giving that direction a value of 0. Lines 60–130 seem simple enough, requesting each direction one at a time and writing each of them into the file, much in the same way we did our descriptive information above. Finally, in line 140 we "CLOSE 1", which puts the file literally onto the disk, and CLOSEs off memory buffer number 1, which we've been using to store the data. And now, if you had done all this, entering data as requested for room number 1, if you were to request a DIRectory of programs you would see the file R1 as one of the programs on the disk in drive number 1. The description of room number 1 would be permanently etched on the disk.

There's more than one room in any adventure, of course, so how do we save the descriptions of the rest of them? The way I do it is lazy but effective. I enter one room, and when I'm finished, I type EDIT 10, which gives me line 10 of ''RMAKER'' on the screen. I then simply go in and change the ''R1:1'' to ''R2:1'' or whatever and then run it again, going on like this until I have all the rooms on the disk. And these are the room descriptions I came up with for ''Space Derelict!'':

ROOM 1
I am hanging on to a hand hold at the only entrance to the ship.
An open airlock leads beyond my line of sight.

ROOM 2
I am now in the airlock. There is a door here that apparently leads into the ship.
Next to the door is a panel of three buttons, blue, black, and green.

ROOM 3
I am in what is apparently an empty room.
S=1: E=1: W=1

ROOM 4
I am in a very large room. There is some sort of pentangular electronic device in the center, but I can't figure out what it is.
Toward the eastern wall there is a very large hologram.
S=1: W=1

ROOM 5
I am in a large room with a pedestal in the center. Some sort of destructive device seems to be planted in the ceiling.
N=1

ROOM 6
This large room has no distinguishing characteristics.
N=1: S=1: E=1

ROOM 7
This room is quite large. In the center there is some sort of pedestal.
Next to the door there are three buttons, blue, black, and green.
N=1

ROOM 8
I am in a long passageway. Something resembling footprints is visible in the dust leading north.
N=1: S=1

ROOM 9
I have reached a crossroads in the passageway. There are footprints (?) leading west.
S=1: E=1: W=1

ROOM 10
This is the end of the passageway. The footprints lead to the western door.
N=1: S=1: E=1: W=1

ROOM 11
I am in a very small passageway leading to a room similar to the last one.
There is a twisted, destroyed metal object in the southern room.
N=1: S=1

ROOMS 12 AND 19
This room seems to be a crew's quarters. Numerous pentangular objects that I presume to be beds.
There is some storage space here, but it has been cleared of all objects.
N=1

ROOMS 13 AND 14
This would seem to be an engine room.
There is the usual configuration of accelerator tubes associated with the Krantz/Warren hyperspace drive, although this is obviously just a variant of the K/W drive.
IF R = 13 THEN E=1; IF R= 14 THEN W=1

ROOM 15
I am in what has to be the main control room. To the north there is a navigation panel with numerous buttons. Behind this are a few rows of pentangular chairs, each with its own computer console.
Toward the southern end of the room is another pedestal, apparently unconnected to the control functions of the ship.
There are four doors here, two on the eastern side and two on the western side. They are paired to the north and to the south.

ROOM 16
I am in a passageway terminating at an eastern doorway.
E=1: W=1

ROOM 17
I am in a passageway.
E=1: W=1

ROOM 18
I am in a passageway. There are doors north and south.
N=1: S=1: E=1: W=1

ROOM 20
The end of the passageway.
W=1

ROOM 21
I am in what seems to be a kitchen.

There are various steel alloy tables, a scullery, convection ovens, and a large open and empty cupboard.
N=1: S=1

ROOM 22
This area seems to be an exercise room.
There is a marked-off ball(?) court, and a large swimming(?) pool, frozen solid.
N=1: S=1

ROOM 23
This is an enormous room stretching all the way to what must be the front of the ship.
There are three doors, one to the north, one to the south (to the west), and the other to the south (to the east).
N=1

ROOM 24
I am in a dramatically colorful triangular crystalline room. At the northern corner is a control panel with a slot in it.
S=1

Getting the information out of the files we've created is even easier than putting it in. We'll talk about how it's done specifically in ''Space Derelict!'' later in the book, but this is a taste of things to come.

```
10   OPEN ''I'', 1, ''R1''
20   FOR X = 1 TO 3
30   INPUT# 1, R$(X)
40   PRINT R$(X)
50   NEXT
60   INPUT # 1, N
70   INPUT# 1, S
80   INPUT#1, E
90   INPUT# 1, W
100  PRINT N
110  PRINT S
120  PRINT E
130  PRINT W
140  CLOSE 1
```

Run this program for each individual file you've made with ''RMAKER'', and you'll get a full replay of everything you've put in. Line 10 is identical to line 10 here in ''RMAKER'' except that now it's OPEN ''I'' instead of OPEN ''O''. The I stands for INPUT and tells the machine that we're going to be taking information from the disk rather than putting it on. Then we INPUT#, as compared to WRITE# in ''RMAKER'', without any need to worry about the commas or what-have-you because once the lines have been LINE INPUT they'll be automatically ''line output.'' The PRINT statements here merely put the data on the screen. If there are errors in any of your descriptions, just run ‘‘RMAKER’’ again for that room, this time getting it right.

And we’ve now got all our rooms laid out.

SWEATY PALMS AT THE KEYBOARD

As I said earlier, I don’t find it necessary to write out all the lines of an entire adventure before programming it now that I have the basic structural module to work from. In fact, after I design an adventure with maps and lists of objects and rooms and so forth, I don’t write anything else on paper. My procedure is exactly the one I am laying out here. First I make up a list of the movable objects and enter that list with locations: I give names to all the CO$(X) variables, and I give numbers to all the CO(X) variables. Then I enter these program lines somewhere around 2000 (we’ll get into why later), and save them as ‘‘INTRO’’. Then I begin with the rooms, making all the files using ‘‘RMAKER’’. Then—and this is crucial—I run off a printout of what I’ve done. It would be practically impossible to design the conversation segment of the program without the room layouts in hand. Most of what the player is going to ask you about is the stuff you’ve described in your rooms, and your program has to be prepared to answer all those questions. You can’t describe a room as having a bearskin rug in it and then not have an answer in your program to the requests LOOK RUG or GO RUG or TAKE RUG. You could wait until you’re in the debugging phase to handle this sort of thing, but why make life harder on yourself? Do it now, in the words of the prophet.

3 SETTING THE SUBROUTINES

THE INTRODUCTION

There are two things you have to do at the outset when the player sits down at the computer to play your game. First you have to tell the player what the game is all about, and then you have to tell the computer what the game is all about. It's a lot easier to explain it to the player than to the computer.

The introduction to the game is the first thing that the player will see when he runs the program, and it should fulfill two purposes. First, it should explain briefly what adventuring is all about, in case you're dealing with a completely novice player. Second, it should set the scene for your adventure, just as an opening chapter sets the scene in a novel. You could just drop your player into the middle of your universe without any warning, but it's a lot nicer to lead him in slowly and deliberately. And there is always the possibility of hiding a few clues in your introduction. This is the only direct communication you will have with the player before the program takes over, and you might just be able to put it to good use above and beyond the call of introducing.

In explaining the play of the game you give the player some idea of how to conduct his end of the conversation with the computer. You describe how the computer understands various one- and two-word commands and perhaps give a few of the more common examples (TAKE, GO, SAVE GAME, etc.). If you're incorporating any special features into the game (using special abbreviations for frequent commands such as INV for INVENTORY or that sort of thing), this is the place to explain them. But don't get too carried away telling the player what to do. Half the fun of adventuring is figuring out what was going on in the mind of the game designer, so you don't want to make your mind an open book.

The other half of the introduction is setting the scene. This is your chance to begin getting really dramatic. You want to pull your player in, tease him, get him interested, and most of all, make the game that follows *real*. Unfortunately, this is a part of the game a lot of designers overlook.

Seeing how the scene was set in ''Space Derelict!'' will give you some idea of how you might do it yourself.

''Space Derelict!''

```
10    REM   INTRO
20    DIM A$(23),CO$(16),CO(16)
30    DIM D(17)
40    COMMONA$(),CO$(),CO(),D(),L,WT,R,AL
50    CLS
60    GOSUB 15000
70    CLS
80    GOSUB 2000
    : REM   SET DIMS
90    INPUT "Do you wish to play a saved ga
        me";C$
100   IF C$ = "Y" OR C$ = "YES" THEN
        GOTO 31000
110   IF C$ = "N" OR C$ = "NO" THEN 130
120   GOTO 90
130   CLS
140   PRINT CHR$ (15)
150   FOR X = 1 TO 10
    : SOUND7,0
    : NEXT
160   PRINT "Series R Robot Explorer number
         182-X3 calling computer control. I
         have an uni- dentified flying obj
        ect in radiometry range."
180   PRINT
190   PRINT "Unidentified flying object is
        a five-sided intergalactic spacecra
        ft. UFO does  not respond to my mes
        sages. I am leaving explorer craft
        to examine more closely."
220   PRINT
230   PRINT "I will be turning command over
         to computer control. Please stand
        by during       transfer."
240   CHAIN"CONWCOM",49900,ALL
15000 PRINT STRING$(30,32) + CHR$ (16) + "S
        PACE DERELICT!" + CHR$ (17)
15010 PRINT
15020 PRINT "You are operating a computer s
        tation located on the Sol side of t
        he planet       Pluto. You are the m
        anager of a platoon of Series R Exp
        lorer Robots."
    : PRINT
```

```
15030 PRINT "Series R Robots are programmed
          to travel the vast expanses of ext
         ra-galactic    space, reporting on
         phenomena they encounter."
15040 PRINT
    : PRINT "Under normal circumstances Ser
         ies R Robots are self-operating. Ho
         wever, in       emergencies or unusu
         al situations the control is taken
         over at the computer      station."
15050 PRINT
    : PRINT "Series R Robots respond to a l
         arge number of simple one and two w
         ord commands.  These include GO, TA
         KE, LOOK, SAVE GAME, etc. Please lo
         ck in your CAPS function key."
15060 PRINT
    : PRINT "On depressing the space bar of
          your console you will take over co
         mmand of the   computer control sta
         tion. Carry  on, Lieutenant."
15070 IF INKEY$ < > " " THEN 15070
15080 RETURN
```

Lines 10–60 are the first lines in the program. Line 10 does nothing but tell us where we are in the program. The adventure module doesn't have to be loaded with a lot of REM statements, and it isn't. REMs allow you to make notes to yourself within a program—the computer ignores them—and they are useful in keeping track of what you're doing. They are especially useful when you set your variables, and at the beginning of subroutines. The more complex your program, the more likely you'll need a lot of notes to yourself detailing what you're doing, and hence the more REMs you'll want to use. If other people are going to be looking at your program, REMs become that much more important and should be used that much more liberally. Since I've written a book to explain "Space Derelict!" I haven't REMed it that much.

With the exception of A$(X), which we'll get to in a minute, you've already met up with the variables in lines 20 and 30. Dimensioning, like Christmas shopping, is best done early—this is a common and recommended programming practice. By the way, line 20 shows how you can easily dimension more than one variable on one line by the use of commas after each index. And line 40 shows COMMON in action, with all the variables on one line, and arrays denoted by empty parentheses. The CLS in line 60 clears any previous printing from the screen.

Line 60 is a GOSUB branch to the subroutine at 15000, which we'll look at now. This subroutine is mostly a collection of PRINT statements that primarily explain how to play an adventure but in a small way begin to set the scene as well. Line 15000 simply gets the words "Space Derelict!" at the top of the screen in reverse printing, i.e., black on white instead of the usual white on black. What is going on in this line is explained more completely later on in this chapter in the section on concatenation, but let's look at it now piece by piece anyhow as it opens up a whole bunch of worm cans that make concatenation pale by comparison.

We mentioned ASCII earlier as the standard numeric code by which computers communicate. What this actually means is that everything you see on your keyboard—and literally everything you can *do* with that keyboard—can be translated into a number. For instance, the letter "A" is given ASCII nuimber 65, "B" is 66, and so on. The lower-case letters begin with "a" at 97. This takes care of the alphabet, two sets of 26 characters each for a total of 52. Altogether there are 256 ASCII characters, numbered from 0 to 255 (again, almost everything having to do with computers starts at 0 rather than 1; it's something you learn to live with). The appendix of your manual shows you all the other ASCII characters, and in fact the TRS-80 manages to use more than just 256 by allowing you to switch back and forth between regular and special characters, something we'll be covering more extensively when we get to our poker game. The point now is that anything we can do with the keyboard can be translated into ASCII, and vice versa, and that is what line 15000 is all about.

The first thing we see here is the command STRING$, followed by two numbers in parentheses. The second of these numbers, if you look it up on your ASCII character list, is a blank space, the same as hitting the space bar on your keyboard. What STRING$ does is tell the computer that you want to create a string that contains the ASCII character specified by that second number and that you want that string to be the length specified by the first number. In this case we have STRING$(30,32), which means we are telling the computer that we want a string containing 30 blank spaces—i.e., the ASCII character number 32 printed 30 times. STRING$ can also use an actual string instead of an ASCII code: the literal equivalent of STRING$(30,32) would be STRING$(30," "). This means that all we have done so far in line 15000 is PRINTed 30 blank spaces.

The next thing we see in this line is the command CHR$, which is actually quite similar to STRING$. CHR$(X) takes the X number in the parentheses and turns it into the equivalent ASCII character. So if we said

```
PRINT CHR$(65)
```

the computer would respond by printing

```
A
```

which is ASCII character number 65. Needless to say, this works all the way up and down the line with all the ASCII characters. What we're specifically doing in line 15000 with CHR$(16) is printing what is called a control character, in this case Control P, or as control characters are sometimes abbreviated, CTRL-P. What Control P does is switch the videoscreen printing around from white on black to black on white, which can sometimes be very useful. It especially looks good in program titles, as we're using it here. Just for your information, you should know about the ASC command that is also available, this one being the opposite of CHR$. Whereas CHR$(65)="A", ASC("A")= 65. CHR$ turns numbers into strings, while ASC turns string characters into numbers. ASC is not used anywhere in "Space Derelict!".

In line 15000 so far we've printed 30 spaces and turned on the reverse printing mode. The next thing we do is easy: We PRINT the words "Space

Derelict!''. This is followed by another CHR$ command—vis., CHR$(17)—which is another control character, CTRL-Q, which turns off the reverse printing and resets everything that follows back to white on black.

Use of these commands CHR$ and STRING$ can be very useful in creating attractive screen output, and if you're not familiar with them it's a good idea to experiment a bit until you get the hang of them. I cannot stress the importance of attractive output too strongly. Whenever someone sits down to play a game or to attack similarly any other program you've created, everything they see and do is framed by how well or how badly you've laid out what they're looking at. The best game in the world will be hard put to survive an ugly appearance. We'll talk more about this in Part 2 of this book.

Lines 15020–15060 are self-explanatory. The occasionally odd spacings result from making sure that the text doesn't break in the middle of a word. It is very easy to keep track of this when you are entering a program. When you actually type in a line you are seeing the line printed 80 characters across even as you type it. So all you have to do is keep an eye on the first pair of quotation marks, being careful not to cross through the column that contains them as you do your typing. It will all come up as clean as a whistle as a result.

You can see that these lines in the program set up the concept of the player as Mission Controller of a platoon of robots. And in 15050, there's a brief explanation of how to command those robots, which act as instructions to the player. You'll notice that we request that the player lock in his CAPS key. The reason for this is twofold. First, if all the player's inputs are in capital letters, we don't have to worry about having our program capable of reading both upper- and lower-case inputs. This wouldn't be so hard, actually—we could examine each letter of each input, making it all upper case as the first step of our conversation routine (we would subtract 32 from the ASCII value of any character with a value greater than 96, since the upper-case alphabet begins at 65 and the lower-case alphabet begins at 97). However, this could slow down processing a little bit, and it's no big deal for the player to hit CAPS. More important, however, is that all the conversation on the computer's end will be in normal upper case and lower case. So if the computer doesn't understand something the player has said, we will respond

I don't understand what you mean by the word WHATEVER.

Since the player's input will be all caps against the computer's lower case, it will stand out that much better in these situations.

The sum of the text from 15020 to 15060 roughly fills the screen. What we want to do is allow the player to read it at his own speed, and when he's finished, he will press the space bar to continue. And that is where line 15070 comes in, with the very important command INKEY$.

Do you know how your computer works? Did you ever stop to wonder how the keyboard communicates with the microprocessor? Well, it's all very simple. The keyboard is a whole and complete unit connected to some mysterious little piece of machinery that collects keyboard input, which in turn is connected to the CPU—i.e., the computer itself. The microprocessor, depending on what instructions you give it, interacts with that little piece of collection machinery in various ways. Usually that little machine collects a number of keyboard strokes and just sits on them. When you eventually hit Enter, the microprocessor takes and does something with those collected keystrokes. What we're going to do here is manipulate that interaction of little machine and microprocessor to our own advantage. INKEY$ is how we do that. You see, whenever we use the command INKEY$ the microprocessor simply reads the most recent character put into that little collection machine and goes on from there without stopping processing. What we're looking for in 15070 is for the last character input to have been a space, " ", which is not to be confused with a null string, "". If we don't get a space, then we GOTO 15070, this same line, and look again. And we continue looping to this same line over and over again until the player finally hits the space bar, at which point we proceed on to the next line of the program. In this particular instance we don't use INKEY$ to any particular advantage; in fact, we could have just said "Hit Enter" and then programmed a line

```
INPUT C$
```

where C$ doesn't mean anything in particular, and the processing simply halts until the player hits the Enter key. But in other situations INKEY$ can be quite useful to us in that other processing can be taking place while we're waiting for the player to do something. Imagine that you're writing a program where you want to move graphics characters across the screen, and you want these characters to keep moving even though you're waiting for some sort of player input. Then you could do something like this:

```
7000  IF INKEY$ = " " THEN 3000
7010  Move some graphics characters
7020  Move a few more
7030  Do a little other business
7040  GOTO 7000
```

Here we're waiting for the player to hit the space bar. If he doesn't, then the activities at 7010–7030 take place, followed by a trip back to 7000 from 7040. If he still hasn't hit a space, the activities take place again. Given the lightning speed with which microprocessors act, you could loop through these lines an incredible number of times before the space bar criterion is finally met and you branch out to the imaginary 3000, where something else is happening. Your INKEY$ function is one of the most important available to you. Take some time to study it in the manual and experiment with it. You'll find yourself using it all the time once you get used to it.

Finally, line 15080 is a RETURN, which marks the end of our subroutine, and brings us back to the line following line 60, whence we GOSUBed in the first place.

The subroutine branch from line 80, which sets the variables, will be covered in the next section, on housekeeping. Line 90 gives the player the opportunity to retrieve an old game to take up where he left off. Here you see the C$ variable; this is the one I usually employ to cover miscellaneous input from the user no matter what the program (AN$ as in ANswer is a common variation of this). It's a good idea to keep some consistency in what you do from one program to another to make your own work easier. If a certain type of variable is always represented by a particular symbol, you'll always know what you're talking about no matter what program you're working on. With so many *really* variable items in any given program, it's nice to have a few variables that remain constant.

If a player does wish to play a saved game, line 100 will branch to the routine at 31000, a sequential-access file routine that will be discussed later in this chapter. This routine fills in the saved values for all the game variables, thereby allowing the player to take up where he left off. After the 31000 routine is completed, the introduction CHAINs to the main program. Line 110 continues the player through the introduction if he does not wish to play a saved game. Line 120 sends the player back to line 90 if an answer other than "YES", "Y", "NO", or "N" is given. Which leads us into the subject of foolproofing your programs.

No matter how clearly you explain something in a program, no matter how carefully you prompt the user for a particular brand of information, sooner or later some answer is going to be put in other than the information you were asking for. It is absolutely essential any time you request input from the user that you make sure the program can handle it, no matter what the input is.

Since it is impossible to predict every idiosyncracy a user can come up with, you have to figure out all the correct inputs and allow for them, and by process of elimination disallow any others. In an adventure, where we're dealing entirely in string input, we have to have adequate responses for all the right answers, and adequate responses for a number of wrong answers too, and the programming sometimes can become a bit complicated as a result. In strategy games, where a move is either allowable or it isn't, things are a lot simpler.

In the case at hand in lines 90–120, we have a simple situation where we are looking for either a YES or a NO answer. But even here we get a little more elaborate than that, allowing both for the complete words YES and NO as well as the shortened Y and N (it is this sort of touch that makes for a more responsive, "intelligent" game, where the player can concentrate on solving your complications rather than solving your programming lapses). If none of these values is input, then the program branches back to line 90 until the player gets it right.

The rest of the introductory routine continues orienting the player to the game universe, and here I get a bit more dramatic. First, I turn off the cursor in line 140—the cursor is the little blinking dash that follows you around the screen, telling you where you are. I do this because I just don't want to see it. It more than serves its purpose when we're waiting for an input, but now we have bigger fish to fry. We turn it off easily enough by PRINTing the control character CHR$(15). When we want to turn it on again we'll PRINT CHR$(14).

Next, we set up a loop to make a little noise in line 150 (if you need an explanation of FOR . . . NEXT commands, see the Glossary). The existence of a SOUND command may come as a surprise to you. It is interesting that you could go out and spend a few megabucks for a computer and not have the perfectly normal command SOUND listed in the manual (although maybe by the time you're reading this Tandy will have corrected this omission). Anyhow, the correct format for this command is

```
SOUND X, Y
```

where X is a number from 0 to 7 and Y is a number from 0 to 31. X controls the pitch, from 0 (the lowest) to 7 (the highest), while Y controls the duration of the tone, from 0 (the shortest) to 31 (the longest). Running through Xs from 0 to 7 gives you roughly one diatonic major scale. What line 150 does here is play around with one of these notes, repeating it 10 times. This gives a nice computerese feel to what follows. The reason? This is our puppet robot talking, good old Series R Explorer Number 182-X3. He's way out in outer space sending back a message directly to your computer console. Why not put a little spice into it? The message is self-explanatory, and finally line 240 CHAINs us into something called CONWCOM. We've already talked about CHAIN and now we're going to be using this command to link smaller programs into one whole. So far we've been concentrating only on the first small program, which we can call ''INTRO''. We will be continuing to work on INTRO for a little while longer, so if you're keying in what we're doing here, keep SAVEing it as ''INTRO'' in the ASCII mode. CONWCOM is the main program of our module, containing all the CONversation With the COMputer, plus a few other things as well. Here is how the four programs break down:
INTRO
CONWCOM
LOSER
WINNER
What we're doing here is going to be fuzzy for a while, but it will all become clear once it's fully laid out. One of the problems of a big program like this is that it is impossible to explain it all in one breath. Be patient as we put together the seemingly unconnected pieces; the whole picture will be quite obvious once you see it laid out.

HOUSEKEEPING

Almost every program needs someplace off the beaten path where variables can be set without colliding into any of the actual processing. This could be done just about anywhere in a program; as a rule it is best to put this material at the end, since it is a one-access-only routine, and assigning it the high listing numbers won't affect processing speed and also will free those lower listing numbers for your more frequent accesses. You see, whenever you put a branch into a program with either a GOTO or a GOSUB, the computer starts at the first line of the program in order to find it—i.e., if you're at 20000 and you have a command that says GOTO 50000, the computer begins looking for 50000 way back at statement number 1 (in their way, computers are pretty dumb, when you get down to it). If a program does a lot of variable sorting at high program lines, you could be seriously hindering your computing capabilities. This concept of flow and speed is important to keep in mind as your programming becomes more sophisticated; we'll discuss processing speed in more detail in Part 2 of this book.

In adventures we have a different problem with our housekeeping other than finding a home for it out of the way of the main body of the program. The reason we've broken down our game into smaller modules is because of memory space limitations. As you've already seen, we set aside a lot of dimension space for our two strings A$() and CO$(). So the question becomes, how does the machine memory store these strings? And is there some way we can use this storage to our advantage?

We've already done some of our housekeeping in lines 2260–2410, when we defined in the program the values of CO$(1 through 16) and CO(1 through 16). When we did this, the computer stored all the contents of those CO$ strings in an area of memory specially set aside for string storage. And as we create more string variables, the computer will similarly store them in that special storage area. This means, when you think about it, that after we initialize our variables they now exist in memory in two different places: in the program where we initialized them, and in the string storage area, where the computer has them available for its own purposes. This further means that we have no need of them in our program listing because the computer will never go back there again, because once it knows the strings, it knows them. So if we could somehow delete the lines that performed the initialization once that information was already stored in memory, we could use that precious programming space for something else entirely. This is the whole point of our CHAINing out of the INTRO routine. Once these variables are set we don't need INTRO anymore. So we get rid of it in favor of other programming we do need.

The variables in ''Space Derelict!'' take up about 45 lines or so. For the most part these variables are radically different from game to game, depending, for instance, on what your movable objects (CO$(X)s) actually are. So let's go right into the specifics, which will set you up for your own parallels.

''Space Derelict!''

```
2000  REM   SET VARIABLES, HOUSEKEEPING
2010  R = 1
2020  REM   A$=ANSWERS FROM COMPUTER
2030  A$(1) = "Instruction not acceptable.
        Please clarify."
2040  A$(2) = "I'm sorry, but I cannot resp
        ond to the  command "
2050  A$(3) = "Okay."
2060  A$(4) = "I cannot go in that directio
        n."
2070  A$(5) = "There is a locked door in th
        at direction."
2080  A$(6) = "I don't have the facilities
        to do that at this time."
2090  A$(7) = "That item is not here at thi
        s time."
2100  A$(8) = "Tell me how."
2110  A$(9) = "I'm not quite sure what you
        mean. Could you be more specific?"
2120  A$(10) = "It seems to be some sort of
         machine part or similar item. It f
        its easily in my  hand."
```

```
2130  A$(11) = "It is the size of a matchbo
        ok. It looks very much like a child
        's clicking toy."
2140  A$(12) = "It is about five inches acr
        oss. The light plays around in it a
        s if it has a life ofits own."
2150  A$(13) = "It is a non-earth origin ph
        aser gun. It is not five-sided. It
        appears to be non-functional."
2160  A$(14) = "It is silver. It opens easi
        ly. It is about seven inches across
        ."
2170  A$(15) = "It is the crystalline struc
        ture of a substance not known on ea
        rth. It is about  7 inches end to e
        nd."
2180  A$(16) = "It's the standard issue Ser
        ies R flash."
2190  A$(17) = "It is about a foot long wit
        h a steel cutting edge of about 8 i
        nches."
2200  A$(18) = "It appears to be made of an
         aluminum-type alloy."
2210  A$(19) = " It is roughly humanoid in
        shape. It appears to have been blas
        ted by some sort ofheat weapon."
2220  A$(20) = "It is somewhat spiderlike a
        nd definitely lethal."
2230  A$(21) = "Nothing happens."
2240  A$(22) = "I don't see anything remark
        able."
2250  A$(23) = "It's the standard Colt lase
        r issued to all Series R Robots."
2420  L = 0
    : REM   L IS FLASHLIGHT, L=1 MEANS ON
2430  WT = 2
    : REM    LIMIT AMT OF INVENTORY TO 5
2440  REM  FOR N = 1 TO 17 ALL D(N)S = 0
2450  REM   AL IS AIRLOCK,0 CLOSED, 1 OPEN
2460  RETURN
```

Line 2010 sets R at 1 so that when we hit the room-layout routine the first time around we'll know where we're supposed to go. Line 2020 explains what A$ is all about (you'll recall that we DIM'd A$ for 23 values back in line 20)—namely, the computer's responses to the players' input. Almost the entire content of an adventure game is conversation with the computer, and in the CONWCOM routine, which we'll discuss in the next chapter, the program attempts to predict all the possible vocabulary input from the player. The conversation routine also directs the computer to print responses to the player's input. If we were so inclined, in the conversation routine we could print out the entire response to each and every player input in each and every instance, but it is a lot easier—and saves a lot of program space—

if we have a virtually complete set of answers already made, so that all we have to do is type in a variable response. It is a lot easier to type:
PRINT A$(6)
100 times than it is to type:
PRINT "I DON'T HAVE THE FACILITIES TO DO THAT AT THIS TIME"
100 times. We'll postpone a complete discussion of A$ until we get to Chapter 4, "Conversation with the computer," but you might as well enter these listings now that they're right in front of you.

Lines 2260–2410 were covered in Chapter 1 and are not repeated here. These were the assigning of the strings to the CO$(X) variables and the placing of the CO(X)s.

Line 2420 is self-explanatory. If the flashlight is on, L has a value of 1; if it's off, L has a value of 0. The need for a flashlight comes up in room 23, where it is too dark to see without it. The need for this line of text is a little less obvious. If no value is originally assigned to a variable, the computer will automatically assign to that variable a default value of 0. So I don't really need to set L as 0 here, since the computer will do it for me the first time my unvalued L pops up. The point is that you might find it a good idea to set *all* your variables during housekeeping time, even if their values are null. Line 2420 could have been done without the L=0 part, but it still would have been virtually imperative to leave in the REM. This way you know that when you look at your housekeeping routine there will be a mention of each and every variable regardless of its value. This is extremely important when it comes to putting together the game-saving routine (and also if you want to sell a program—software publishers like lists of variables in the documentation, and if you have variables in your program that aren't in your documentation . . . good-bye, Charlie).

WT in line 2430 stands for weight. This is how we limit the inventory. If we wanted we could have no such limitation and allow our puppet robot to carry everything he can get his hands on, but this is both unrealistic and a little too easy. In "Space Derelict!" there is a weight limitation of five items (the actual limiting of the weight takes place in the conversation-with-the-computer segment). If you wanted, you could get much more complex than this. You could assign a literal weight to each movable object and then limit your puppet to a certain poundage (there might be some real purpose to this in a Dungeons and Dragons–type adventure). You might want to experiment with different weight limitations other than five objects when you actually start debugging your own adventure (five is an awfully small number in many situations, but again you have to be careful not to make it too easy). In any case, the value of WT=2 here refers to the two objects our puppet robot is already carrying, the flashlight and the laser pistol, set in lines 2320 and 2330.

Line 2440 simply reminds us that all the doors are closed. We already discussed what the variable D was all about and how a value of 0 means closed door and a value of 1 means open door. As we said a moment ago, it is not necessary literally to set all these values, since they are all 0, but once again it is important to have a mention somewhere in the housekeeping of D(X) and what it is all about.

Line 2450 is similar to 2420 and 2440, again without the superfluous assignment of the 0 value. This line is self-explanatory and refers to the airlock door. And line 2460 RETURNs us back to line 80, whence we came.

CONCATENATION

Concatenation is the biggest word in the computer vocabulary and one of the simplest concepts. But unfortunately most computer writers wreak havoc with it, blowing it way out of all proportion to its simplicity. Since we're going to be concatenating up a storm in both of the game programs in this book, we might as well go over this well-trodden ground a bit ourselves.

Simply put, concatenation is the stringing together of two or more strings, which we usually do with the "+" symbol. If A$="FRED" and B$="J. MUGGS", and if we were to enter a command:

```
C$ = A$ + B$: PRINT C$
```

the computer would respond with:

```
FREDJ. MUGGS
```

which is the sum of the two strings A$ and B$. If we were to do it the other way around and say:

```
C$ = B$ + A$: PRINT C$
```

the computer would respond with:

```
J. MUGGSFRED
```

The computer doesn't care what's in your strings; when you concatenate them it adds them together, and you can do whatever you want with the results.

Let's go back to:

```
C$ = A$ + B$
```

where A$="FRED" and B$="J. MUGGS". This gives us a C$ of "FREDJ. MUGGS". The nice thing about string variables is that we can make them dance for us once we know how. We can turn strings into numbers, numbers into strings, read strings left to right and right to left or from the middle out—you name it, we can do it. And in programming games, sooner or later we will.

Let's assume that you are a good grammarian but not much of a trivia buff. Looking at C$ you realize right away that a space is missing between FRED and J. Let's add the missing space by using our dancing string commands. First, enter:

```
C$ = LEFT$ (C$,4) + " " + MID$ (C$,5)
```

Then enter:

```
PRINT C$
```

in order to get:

```
FRED J. MUGGS
```

If you know how that worked you can skip along to the next section. If you don't, settle in.

We can read any given string from right to left, from left to right, or from anywhere in the middle to the end. The commands we use to do this are LEFT$, RIGHT$, and MID$. The LEFT$ and RIGHT$ commands are the mirror images of each other: RIGHT$ counts off from the right, and LEFT$

counts off from the left. So the LEFT$(C$,4) that we used above took the string C$ and counted off four characters from the left. LEFT$(C$,3) would have given us ‘‘FRE’’, LEFT$(C$,2) would have given us ‘‘FR’’, and LEFT$(C$,8) would have given us ‘‘FREDJ. M’’. As long as the number is no greater than the total length of the string, we get the number of characters we ask for counting off from the left.

RIGHT$ works the same way from the right. RIGHT$(C$,4) would have given us ‘‘UGGS’’. As a matter of fact, we could have added our missing space in the following way:

```
C$ = LEFT$ (C$,4) + " " + RIGHT$ (C$,8)
```

since RIGHT$(C$,8)=‘‘J. MUGGS’’. Keep in mind that the space we added is, in effect, another string, a string one character long, that character being a space. This is *not* the same as a null string, which is no string at all, or a string of zero length (‘‘’’).

You’ll notice, though, that instead of originally using RIGHT$ to add the space I used the third string-reading command at our disposal, MID$. MID$ works from the middle, and it works in two ways, Let’s look at C$ again as we’ve revised it:

```
C$ = "FRED J. MUGGS"
```

Let’s say that this time we are true trivia buffs who can recognize that we have misnamed the chimpanzee who used to drive Dave Garroway crazy. That chimp’s name was J. FRED MUGGS. Using MID$ we can adjust C$ accordingly:

```
C$ = MID$ (C$,6,3) + LEFT$ (C$,5) + RIGHT$ (C$,5)
```

If we were now to type:

```
PRINT C$
```

we would get:

```
J. FRED MUGGS
```

which is the correct and 100 percent bona fide name of the primate in question. How did we do that? MID$ reads a string from a specific given character for as many characters as we indicate in the string from left to right. So MID$(C$,6,3) reads C$ for three characters starting at the sixth character. Looking at the corrected C$, MID$(C$,4,4) would be equal to ‘‘FRED’’, which is the four characters staring at the fourth character. MID$(C$,10,5) would give us ‘‘MUGGS’’, which is the five characters beginning at the tenth character. But note also that MID$(C$,10) would *also* give us ‘‘MUGGS’’. When no second number is given to tell the computer how many characters to count off, the computer simply counts off to the end. This variation of MID$ has its uses too, and although it may look deceptively similar to the RIGHT$ command, there are times when the two will not be interchangeable, so you need both of them in your little bag of tricks.

Let’s get into some muddier water. The computer also can turn numbers into strings and, on occasion, turn strings into numbers. And these too will be important concepts in game programming. Let’s try a new formula:

```
H$ ="P23"
```

This formidable little item is quite similar to what we’ll be working with later on, in our poker game. Let’s further say that:

```
X = 2: Y = 3: Z = 4
```

Now we have three numeric variables, X, Y, and Z, and one string, H$.

Let's say now that we want to change H$ and add to it the number 4 (the value of Z) at the end to get P234. There's an easy way to do that with the information at hand plus one new command, STR$ (not to be confused with STRING$):

```
H$ = H$ + STR$(Z)
```

STR$ takes the value of Z and translates it into a string. STR$ literally puts quotation marks around numbers. Since Z=4 then STR$(Z)="4". Now when we enter:

```
PRINT H$
```

we will get what we wanted, which is "H234". We can also work this the other way around. Let's say we want to find out the numeric value of part of a string. If we were to enter:

```
PRINT VAL (RIGHT$ (H$, 1) )
```

we would get the response:

```
4
```

VAL is the opposite of STR$. VAL turns strings into numbers, provided that string or part of a string is a number to begin with.

```
PRINT VAL (RIGHT$ (H$, 3) )
```

would give us:

```
234
```

as an answer. But:

```
PRINT VAL (LEFT$ (H$, 1) )
```

would get us into trouble, because LEFT$(H$,1) is the letter H, not a number, and therefore a valueless commodity as far as the VAL command is concerned.

This is not to say that the computer is ignorant of the ranks of letters in the alphabet, however. The TRS-80 can compare alphabetical values if you so direct it. As far as the TRS-80 is concerned, "A"<"B" and "B"<"C" and so forth down the line. Also, "BLACK"<"BOTTOM", "CLARK"<"KENT", again all the way down the line. So if you ever want to sort a list of names alphabetically, the computer can do it for you.

There is one last string command worth knowing, LEN. LEN measures the length of a string.

```
PRINT LEN (C$)
```

would give us a response of:

```
13
```

because the character length of "J. FRED MUGGS" is 13. LEN can be handled just like any other number—that is, the command:

```
PRINT LEN (C$) * 3
```

would give you a response of:

```
39
```

which is 13 multiplied by 3. The VAL command can also be used in this way, as in:

```
PRINT VAL (RIGHT$ (H$, 4) * 3
```

which would give us an answer of 4 times 3, or 12.

The last important thing to mention about strings—and don't worry if you're still a little fuzzy on them because you'll see all these commands in action before this book is over—is that concatenation works only in one direction. There is no such thing as negative concatenation. Let's say that:

```
P$ = "MOTHER": Q$ = "HER"
```

If you were to try to get the word "MOT" in the following fashion, you would get a syntax error:

```
PRINT P$ - Q$
```

For some reason this seemingly logical handling of strings is unacceptable, even though it appears no more arcane then P$ + Q$. And there are times when a subtraction of strings would really fill the bill in what you are doing. But you're stuck without it. Never fear, though, because the commands you *do* have will do everything you need, provided you organize them correctly.

SIMPLE FILE ROUTINES FOR SAVING GAMES

One of the nicest things about adventure games is the player's ability to save a game whenever he wants to. The next time he sits down and decides to attack it again he can indicate that he wants to play a saved game, and *voilà!* He is right back where he left off. This saves a lot of wear and tear roaming about in the same old dungeons for the umpteenth time when he's already scoured through them quite successfully, and it is surprisingly simple on the part of the programmer to include this feature in a game.

A game is saved on a disk via the agency of a sequential-access disk file similar to the ones used in creating our room descriptions. This final routine in our INTRO program is the one that allows the player to take up in the middle of a saved game. Unfortunately, the routine where a game is actually saved occurs in the CONWCOM conversation segment. This means that we're going to jump ahead a little bit into CONWCOM in order to understand INTRO completely. Since we've already covered disk files pretty thoroughly, this shouldn't be too much of a problem.

In our adventures we have a finite number of variables—R, the CO(X)s, the D(X)s, and so forth. We set them all and named them in the housekeeping routines, and along the way as a game progresses some of their values may have changed, but still they are the same in number, and more to the point, these variables and these variables alone are responsible for controlling the game. The values of these variables tell us what room we are in, what we are carrying, what doors are open and what ones are closed, what items are lying around where—literally everything there is to know about the game. So if we collect all these variables at any given point and save them, and then go back later and read them out, they will tell us exactly where we are within a game. Furthermore, by putting them into a disk file we etch them onto a disk and store them away—that is, their values won't disappear when we turn off the machine. In other words, we can save a game in midstream and go back to it whenever we wish.

The trick, of course, is to write a file and read it in exactly the same sequence, thereby giving meaning to what would otherwise be a meaningless set of numbers. Let's take a look at the game-saving routines in "Space Derelict!"

```
30000 REM  SAVE GAME
30010 OPEN"O",1,"SDFILE:1"
30020 WRITE#1,R
30030 FOR X = 1 TO 16
30040 WRITE#1,CO(X)
30050 NEXT
30060 WRITE#1,L
30070 WRITE#1,WT
30080 FOR X = 1 TO 17
30090 WRITE#1,D(X)
30100 NEXT
30110 WRITE#1,AL
30120 CLOSE1
30130 PRINT "Game saved."
30140 GOTO 3000
31000 REM   REOPEN SAVED GAME
31010 OPEN"I",1,"SDFILE"
31020 INPUT #1,R
31030 FOR X = 1 TO 16
31040 INPUT #1,CO(X)
31050 NEXT
31060 INPUT #1,L
31070 INPUT #1,WT
31080 FOR X = 1 TO 17
31090 INPUT #1,D(X)
31100 NEXT
31110 INPUT #1,AL
31120 CLOSE1
31130 PRINT "Please stand by..."
31140 CHAIN"CONWCOM",49900,ALL
```

Lines 30000–30140 contain the routine for saving a game, which is handled in CONWCOM, and lines 31000–31140 contain the routine for replaying a saved game, handled in INTRO. If you compare lines 30000–30120 to lines 31000–31120 you will find that they refer to exactly the same variables in exactly the same order. And as you also can see, they include all the variables with changeable values. They do not include, for instance, the string values of the CO$(X)s, because these values do not change. CO$(1) is ODD PIECE OF METAL and it *always* is ODD PIECE OF METAL regardless of where it is, and the value of the CO$(X)s will be set in the housekeeping routine at 2000 and stored in memory at the outset and need not be saved here. The only technical differences between the file that saves the game and the file that replays the game are the use of OPEN "O" for Output when we save versus "I" for Input when we replay, and the WRITE command versus the INPUT command. When we've finished reopening this file we tell the player to wait a minute in line 31130, and then we CHAIN into CONWCOM exactly as we did in line 240 when we didn't play a saved game. Also note in line 30130 that a game can be saved at any point and does not necessarily mean that the game has ended, which is why the program branches in this fashion.

If your adventure has more or different variables than those here you can simply tack them onto this basic structure both for INPUTing and WRITE-ing. The important thing is that the INPUT statements and the WRITE statements reflect each other in precise order: You don't want to input CO(X) when you think you're inputting values of D(X). Your program won't know the difference, but the player certainly will when he tries to proceed with the game. And always adhere strictly to the various OPEN and CLOSE commands or else be prepared for real disaster.

When passing your game along to others, it is a good idea to create a saved-game file and put it on the disk along with the game in case some wise guy tries to play a saved game his first time through (line 100). Simply start up a game and get yourself into room 1, where you SAVE GAME and that's the end of it. That way your wise guy won't bomb the program by trying to read a nonexistent file, and you won't give away anything he wouldn't have found out for himself anyway two minutes later.

And now we've covered all of INTRO, the first small program in the adventure module. Let's just go over it one more time to make sure you've got it all. Lines 10 through 240 were the main section, in which we began by dimensioning our arrays. From this main section we branched to the title and game description at 15000, and then to the housekeeping/initialization at 2000. Then we offered our player the option of picking up a saved game at 31000, or starting at the beginning. In either case we finished up by CHAINing to CONWCOM starting at 49900 and carrying the values of ALL the variables. And that is exactly how you'll do it for any adventure, with the same routines at the same lines, ready to hold whatever variables you want to put into them. These line numbers were chosen (why are we all the way up at 15000, for instance?) so as not to conflict with the lines of the other programs we're CHAINing to. It doesn't matter to the computer whether they're the same lines or not, because the CHAINed program erases the one already in memory, but it makes our job that much easier if we keep those line numbers separated so we can keep track of what we're doing.

WINNING AND LOSING

The next two small programs contain scenarios for winning and for losing the game. These two are actually short enough to have fit into CONWCOM, the main module, but there is no real need to do so because they do break out of the main flow, and it is not inconceivable that you could make them much more elaborate, and as a result, too long to fit comfortably. We CHAIN to these from within CONWCOM.

There is usually only one way to win any adventure: In a treasure hunt you have to collect all the treasure, and in a situational adventure you have to reach the end of the situation. In "Space Derelict!" the insertion of the filled box into the slot in room 24 signals winning the game, and as soon as this action is completed, the program branches to line 17000, which then CHAINS into the winning scenario (which happens to be a program called WINNER). Any situational adventure will have a similar set of circumstances that signal success and a similar branch to the winner's circle.

Treasure-hunt adventures require some other apparatus to signal their successful completion. Usually you have to collect a certain number of treasures and then deposit them somewhere for safekeeping, generally some neutral room in the universe. An easy way to measure how the player is doing is to count the number of treasures stored in the treasure room every time the player adds a new item to the collected loot. Let's say there are 20 treasures, and the storage room is number 32. This loop would figure out whether or not the player had won:

```
FOR X = 1 TO 20
IF CO(X) <> 132 THEN X = 20: WINFLAG=1
NEXT X
IF WINFLAG <> 1 THEN GOTO 17000: REM WINNER'S CIRCLE
```

This routine accounts for each movable object. If any one of them is not in room 32, the loop is aborted and a flag is set at 1, but if all 20 are there, you branch to the winner's circle.

In ''Space Derelict!'' the winning scenario looks like this:

''Space Derelict!''

```
17000 PRINT "There's a burst of light..."
    : GOSUB 63000
17010 PRINT "An explosion of color..."
    : GOSUB 63000
17020 PRINT "A roar of thunder..."
    : GOSUB 63000
17030 PRINT "And another message from the
       ship, which I will now transmit. P
       lease stand by."
    : GOSUB 63000
    : GOSUB 63000
17040 PRINT
17050 PRINT CHR$ (16)
17060 PRINT "CONGRATULATIONS, PEOPLE OF TH
       E SYSTEM OF THE NINE PLANETS. YOU
       HAVE MANAGED TO  DEFUSE THE SHIP.
       IT SURPRISES US THAT YOU WERE CAPA
       BLE OF SUCH CLEVER THINKING."
    : PRINT
17070 PRINT "BUT BEWARE... THE KRENN WILL
       BE BACK AGAIN SOME DAY. AND NEXT T
       IME IT WILL NOT  BE SO EASY FOR YO
       U."
    : PRINT
17080 PRINT "FAREWELL, PEOPLE OF THE SYSTE
       M OF THE NINE PLANETS."
    : PRINT
    : PRINT
    : PRINT
17090 PRINT CHR$ (17)
```

```
17100 PRINT "END TRANSMISSION"
17110 END
63000 FOR PAUSE = 1 TO 2500
    : NEXT
    : RETURN
```

There isn't much to winning other than satisfaction. Lines 17000–17020 build up a little suspense—that way the player knows something unusual is happening. Then once again we print the control character that gives us reverse printing of black on white (line 17050), and then we congratulate the player, but it's just the sort of backhanded congratulation you would expect from the Krenn, with their threat of returning—as if they hadn't caused enough mischief already. Line 17110 ENDs the game once and for all.

A losing scenario assumes that the player has stepped into one of your deadlier traps from which there is no return. When this happens it is advisable not to be too snide about it; sooner or later the player *will* conquer your game, and he who laughs last . . . . Be nice to your audience; your player is doing you the supreme honor of spending his time with your game. So when the player loses, don't program a message such as:
PRINT "YOU LOSE, STUPID!"
That's just not good computer etiquette. But then again, the player did screw up, so you don't have to bend over backward to congratulate him on his intelligence, either.

Losing in "Space Derelict!" comes as a result of pressing a wrong door-control button, entering room 5 without deactivating the killer device, fighting a losing battle with the robot in room 23, plus one or two other miscellaneous screw-ups. This is what the losing scenario looks like:

"Space Derelict!

```
16000 FOR N = 1 TO 10
16010 SOUND7,0
16020 PRINT "END TRANSMISSION"
16030 NEXT
16040 PRINT
    : PRINT "It looks like the end of the
        Solar System. Too bad..."
    : PRINT
16050 INPUT "Care to try again";C$
16060 IF C$ = "Y" OR C$ = "YES" THEN RUN "
        INTRO"
16070 IF C$ = "N" OR C$ = "NO" THEN END
16080 GOTO 16050
```

This is actually a whole program entitled LOSER, CHAINed to from line 16000 of CONWCOM. Once again I took advantage of my fictional situation of a puppet robot, this time by printing a message that just reeks of computerese. The SOUND command in 16010 is going to give us 10 nasty little buzzes, but it's the message that goes with them that adds that little computer touch—the phrase END TRANSMISSION printed over and over again. The first time the player sees it, it's quite a surprise; after that, it's quite a letdown.

Line 16050 gives the player an opportunity to try his luck again. If he does opt for another shot at it he's dealing from a cold deck, due to the RUN "INTRO" command in 16060. This line erases all the variables in the memory of the computer, as does any RUN or LOAD (or even a CHAIN without a COMMON), so everything has to be reset all over again, which is why we go back to the beginning of INTRO, where once again we DIM this and that endlessly. Of course, once the player does get back to the beginning he has the option of taking up where he left off with a saved game if such is his pleasure, so although you're sending him back to square one in the program, he has his chance to jump ahead if he's stored any games-in-progress in his travels.

4

CONVERSATION WITH THE COMPUTER

THE NATURE OF THE BEAST

A player plays an adventure by sitting down at the computer and literally conversing with the program. The first step for the player is figuring out what language the game speaks. When he finds the right mode of communication, the right combination of words that the program can understand, the player can successfully get the program to respond to his commands. One of the joys of adventuring is that discovery of the extents and limitations of the game's vocabulary. Needless to say, the more extensive that vocabulary is, the more potentially rewarding it is to the player. And conversely, the more limited that vocabulary, the more frustrating the game can become. It is up to you as the game designer to develop as extensive a working vocabularly as you can for each game you create.

Adventures traditionally require one- or two-word inputs from the player, and although there are some games on the market that can handle sentences of more than two words, you're really asking for it if you try to emulate them at this stage of your career. Believe me, two words are enough to keep straight; more than that certainly stretches the capabilities of your average workingman's everyday Basic. The problem for the designer is squeezing in meaningful programming to analyze—or parse—these long sentences quickly and accurately. The work involved is a geometric progression of complexities as the player inputs get longer and longer, and more than two words can make the parsing a real nightmare. After a little practice you might be up to it, but for our purposes, two-word inputs will be plenty. Besides, in many respects the ability of a program to parse long sentences is merely window-dressing. It's quickest and easiest for the player to use the shortest inputs possible.

"Parse" is an interesting word. Parsing is breaking down a sentence into its components. Since in our adventures we are limiting ourselves to two-word sentences, the parsing becomes relatively automatic, since an imperative two-word sentence by definition contains exactly one verb and one noun, in that order. We're asking our player to enter commands—i.e., imperative sentences—so while there are conceivable complete two-word sentences not in the imperative mode, such as DOGS BARK, there is no earthly reason why any player would input such a sentence and expect any sort of response to it. An imperative sentence also can contain only one word, but that one word must be a verb (you wouldn't command someone to FRANKFURTER, for instance). Anyhow, virtually the entire contents of the conversation-with-the-computer segment of the program, called CONWCOM, consist of parsing the player input, analyzing it, and responding to it. Essentially you will be creating a working vocabulary, comparing what the player says to what you have in that vocabulary, then acting on the results.

There are certain words you will always use in every game, such as GO, TAKE, DROP, and LOOK. In fact, these particular four words provide the basic framework for your game vocabularies and will require some of the longest routines in the program. Other words will be more game-specific: If there is no body of water in the game universe, for instance, it is extremely unlikely that the player would come up with the command SWIM, and therefore it's relatively unnecessary for your program to accommodate that word. But if there *is* a body of water, your program should be ready to handle SWIM, WADE, FISH, DRINK, and anything else you can think of that remotely has to do with large amounts of liquid.

The keys here are flexibility and completeness. By this point in your game design you have set up all the factors of play, and you have incorporated all those factors into a universe waiting to be explored. But now you must prepare yourself to communicate to the explorer, and to do so in such a way as to be truly bend-over-backward accommodating. You have to prepare your program for *every* possible player input; more importantly, whenever that input is reasonable, even if the input is not "correct" vis-à-vis your game, you have to respond to it meaningfully. In effect you have to read the mind of every player who will ever sit down with your game.

It is extremely instructive to watch a complete novice attack an adventure. An experienced player becomes pretty good at getting instant communication with a new game, but a novice has really no idea of what's in store for him, so he'll try anything. And some of those tries aren't half bad. While some will be off the syntactical mark a bit (it takes a while to learn that sentences such as GO TREE or UNLIGHT LANTERN are actually meaningful), the vocabulary usage will be most illuminating. Not knowing the accepted modes of adventuring communication, the novice will instead try the more logical ones or else the most completely outrageous ones. This in fact becomes the true test of your adventure: When the player tries something quite logical in context, does your adventure respond just as logically? It should. And if the novice does something completely outrageous, does your adventure respond logically to that outrage, or is the response a non sequitur as silly as the input? If your adventure can't handle both the predictable and the unpredictable, it's not really ready for prime-time players.

One of the things we'll try to do in this chapter is create a catalog of good vocabulary to include in adventures, but that catalog will by no means be complete. Every game you work on will have its own nooks and crannies where special words will be required that are unnecessary in other situations. In "Space Derelict!" for example, I use the words INSERT and MELT, both of which obviously fit very specific criteria here but that are not necessarily ever going to pop up again in other games. The point is, if there's nothing MELTable in the game universe, the player is never going to enter the command MELT MOUSE or whatever because it's just not going to make any sense and will therefore never occur to him. You have to be especially aware of those areas in your game where special, unusual vocabulary is needed and be prepared with as much of it as possible. And as a corollary to this, whenever possible you must be prepared with synonyms for your acceptable words. A moment ago I referred to LOOK, GO, TAKE, and DROP as the backbone words in the adventure vocabulary; so too are their synonyms EXAMINE, N (or E or W or S), GET, and LEAVE. For every verb there are probably two other similar verbs that you ought to include in your adventure. Of course, you can't throw the whole dictionary into the machine, so basic horse sense will limit you eventually, but the more common and likely variations all must go in to make the game really playable. The same holds true for nouns, but less strongly. Let's say, for instance, that a room is described as having a painting, and the player enters the command LOOK PICTURE. Is it necessary to respond to that command? Some would say yes, some would say no. I'm inclined against it myself in most cases, simply because there's enough work to do on the verb side of the statements without killing yourself on the noun side as well. There's no reason why a player can't describe an item using the words already on the screen in front of him. But if your room includes something called CRYSTAL WHALE, then it does make sense that your program be able to understand both the terms WHALE and CRYSTAL. But even here there's some debatability, and given my druthers I'd worry more about other aspects of vocabulary and let the player figure out that certain items have certain names, and if he gets it wrong the first time (by referring to the CRYSTAL, for instance), then he can get it right the second time (by referring to the WHALE), and he'll save us quite a bit of programming work by so doing.

No matter how much vocabulary you use, however, sooner or later your player will come up with something your program can't handle, and you have to be just as prepared with an adequate response here as you were when his command was workable. There are two general sorts of unacceptable player input, and you should respond to each appropriately. First, there is input in which the verb isn't included in your game vocabulary, and the player should be made aware of this in no uncertain terms. If a player enters SLURP SOUP, your program, which does not recognize the word SLURP, should respond with something like, WHAT DO YOU MEAN BY *SLURP*? This shows the player that SLURP is unacceptable but ignores SOUP entirely, leaving that issue for another day. On the other hand, let's say that you do include SLURP in your vocabulary, because slurping the soup is an important step in solving the puzzle, but your player, bumbling around, enters the command SLURP PIANO. Here we have a problem because we do know what slurping is, and we can't say we don't know what PIANO is because there indeed is a piano in your adventure. You can't recognize it one time and not the next. To be on the safe side, when our program recognizes a verb but the noun following it is not one we are expecting, we simply vague it out for all wrong nouns with a response to the effect I CAN'T SLURP [WRONG NOUN]. This tells the player that the action requested is unacceptable but doesn't preclude use of the same vocabulary in other combinations.

The makeup of CONWCOM, the conversation with the computer segment, is not complex, although the segment itself is long (as a rule, the longer this segment the better your game because it can respond to that much more vocabulary). The segment begins by getting the input from the player; then it analyzes that input to determine the length of the first word (or, if it is a one-word command, the length of that one word). Then the program scuttles off to an appropriate section consisting of words of given lengths: All the one-letter first-word commands are grouped together, as are the two-letter commands, the three-letter commands, etc. The computer compares the player input to the words in the appropriate group, and if a match is found, then it follows the instructions at that match and the game will progress accordingly. If no match is found by the end of the group, then the program will respond that the first word of the player input is not one that the computer understands, and the player will be prompted for a new input.

As I've pointed out before (and which is now quite obvious since we've pretty much seen all of the rest of the program), most of the real business of the game is handled in the conversation segment. If a player TAKEs or LEAVEs a movable object, this is where it is done. If a player wants to move the puppet, this segment does the work. If the player wants to look at something, or touch it, or kick it, or eat it, or whatever, this is where the action is. Fortunately, though, none of this is very complicated in programming terms, so while the sheer length of what we'll be doing here may be daunting (even the simplest adventure can eat up 30K of memory without batting an eye), the process of doing it is very simple, so don't be put off by the vastness of it all.

CONWCOM—AN OVERVIEW

The main body of CONWCOM is our conversation with the computer. This programming begins at line 3000 (the actual first line of CONWCOM), stretching all the way to line 7900. All of these lines, totaling roughly 500, contain specific responses for all the possible player inputs. But there are a few other pieces of CONWCOM that we should dispense with right off the top, because they are the embarkation points for all the CHAINing we've been discussing in and out of this central module. We should tie off these loose ends before getting into any new business.

First off, you'll recall that the INTRO program CHAINED into CONWCOM at line 49000, so let's start there with our discussion.

"Space Derelict!"

```
49900 COMMONA$(),CO$(),D(),L,WT,R,AL
50000 ROOM$ = "R" + STR$ (R)
50010 IF MID$ (ROOM$,2,1) = " " THEN ROOM$
        = LEFT$ (ROOM$,1) + MID$ (ROOM$,3)

50020 OPEN"I",1,ROOM$
50030 FOR X = 1 TO 3
50040 INPUT #1,R$(X)
50050 NEXT
50060 INPUT #1,N
50070 INPUT #1,S
50080 INPUT #1,E
50090 INPUT #1,W
50100 CLOSE1
50110 T = 0
    : IF R = 23 THEN 50221
50120 PRINT R$(1)
50130 IF (R = 11 AND CO(11) < > 105) THEN
        50150ELSE IF LEN (R$(2)) > 0 THEN
        PRINT R$(2)
50140 IF LEN (R$(3)) > 0 THEN PRINT R$(3)
50150 GOSUB 13000
50155 PRINT
50160 IF R = 23 THEN 3000
50170 IF N = 1 THEN PRINT "I can go north.
              ";
50180 IF S = 1 THEN PRINT "I can go south.
              ";
50190 IF E = 1 THEN PRINT "I can go east.
              ";
50200 IF W = 1 THEN PRINT "I can go west."

50210 GOTO 3000
50221 IF L = 0 OR CO(8) < > 1 THEN PRINT "
        It's too dark to see in here!"
    : PRINT
50230 IF CO(8) < > 1 THEN 3000
50240 IF L = 1 THEN 50120ELSE GOSUB 63000
50250 PRINT "I'm turning on my flashlight.
        Please stand by."
    : GOSUB 63000
50260 L = 1
    : GOTO 50120
```

The first thing we see in 49900 is the complementary COMMON statement that assures us that all our variable values set in INTRO will be CHAINed over successfully into CONWCOM. It is a good idea to put your COMMON statement at the very beginning of a program where you've CHAINed in, which in this case is line 49900. If we have CHAINed to the start of

CONWCOM we would have put the COMMON as the first line at the start of the program. The rest of this segment of the program, from 50000 to 50260, is concerned with displaying the room descriptions on the screen.

There are two ways we will get to line 50000. The first way, of course, is from 49900, coming in through the INTRO. The second way is from various places within CONWCOM. Whenever a player moves from one room to another, or requests a new description of the room he is in (if he does enough other things in a room the original description can scroll up and away very quickly), we will branch to 50000. And when this happens, the variable R, representing the number of the room the player is in (or has just entered, if he's made a move), is of the utmost importance, because we're going to want to open and read the room file (R1 through R24 in "Space Derelict!") that is correct for that room. So the first thing we have to do is translate R into something TRSDOS can understand. This is done in line 50000, using the STR$ command. You'll remember in our discussion of concatenation that STR$ is the command that turns numbers into strings. So what we're doing in 50000 is creating a new string called ROOM$, which string consists of the letter "R" concatenated to STR$(R). This means that if we're in room 1, then ROOM$ will equal R1, and if we're in room 2, it will be R2, and so forth. But there's a catch to this, which we take care of in 50010. You see, STR$ is set up to allow manipulation of negative numbers. For instance, STR$(− 5) would be printed on your screen

```
−5
```

That's all well and good in most situations, but what the TRS-80 does with positive numbers is save a space for a nonexistent minus sign. So when we try to print STR$(5) we actually get

```
 5
```

printed one character away from the left of the screen. This means that instead of getting a "5" we're actually getting a " + 5", with the plus sign implicit and unexpressed. What happens, therefore, when we create ROOM$ is not that we're getting R1 or R19; we're getting R 1 or R 19, a fact we have to take care of somehow, which we do in 50010. We know that the "R" in our ROOM$ will be okay, and the problem will be the second character in the string, MID$(ROOM$,2,1), which will be a blank (" "). So we delete that blank in this line by concatenating LEFT$(ROOM$,1) with MID$(ROOM$,3), thus giving us the correct file names the way we saved them with RMAKER.

The next thing we do is OPEN the files and read the information out of them exactly the way it was put in, in lines 50020–50100. Then in 50110 we set a new variable T as 0, which you don't have to worry about at this point, and if R = 23 we branch to 50240, which we'll also postpone for a while. Let's stick to the main road before we go off on any tangents.

Having read out the contents of our appropriate ROOM$ file, we now have new correct values of R$(1–3) and N, S, E, and W, for whatever room we're in. The first thing we do is PRINT R$(1) in line 50120. Then, barring one exception in 50130, if the LEN of R$(2) (and in the next line, R$(3)) is greater than 0—i.e., not a null string—we print the contents of that variable. This means that now, exceptions aside, we have printed up to three batches of text for the room we're in, the same three batches we created in RMAKER. The exception of room 11 in 50130 refers to a descriptive line in R11 about a twisted, destroyed metal object being in the southern room (room 11 is the corridor leading from the room with the hologram to the room with the zapper); if we're in room 11, before printing this line we first make sure that that object, CO(11), is indeed in that room (CO(11)=105). As for the new wrinkle of ELSE, let's get the rest of this section under control before going into that one.

The next thing we do, in line 50150, is branch to the subroutine at 13000. This subroutine prints to the screen the contents of a room by measuring the values of all the CO(X) items in the game. If any CO(X) is equal to 100+R, then that CO$(X) is printed. The subroutine at 13000 looks like this:

"Space Derelict!"

```
13000 FOR X = 1 TO 16
13010 IF CO(X) = 100 + R THEN FLAG = 1
    : X = 16
13020 NEXT
13030 IF FLAG = 0 THEN RETURN
13040 PRINT "The following items are in th
       is area:"
13050 FOR X = 1 TO 16
13060 IF CO(X) = 100 + R THEN PRINT CO$(X)
       ;
13070 NEXT
13080 FLAG = 0
13090 RETURN
```

This is a relatively simple sort program. We take all our movable objects from CO(1) through CO(16) and measure their value in line 13010. If any of them is equal to 100+R, that means there is at least one movable object lying around. We set a flag with a value of 1 and augment X to its maximum value of 16—once we know we have a movable object lying around, we know we have more to do, and there's no point in measuring any more CO(X)s, whereas if we go through all 16 values without getting a hit, we simply RETURN in line 13030. If we have a hit, we print the line in 13040 followed by the correct CO$(X)s corresponding to the CO(X)s equal to 100+R. And after all the CO$(X)s are printed, the FLAG is reset at 0, and we RETURN whence we came, line 50150.

Again barring the exception of room 23, the next thing we do in our 50000 room display routine is show the exits. This is done in 50180–50210. If any of our N/S/E/W variables is equal to 1, then "I can go WHEREVER" is printed on the screen (note the semicolon at the ends of these lines, allowing all this information to be printed on one line). Then finally we GOTO 3000, which is the beginning of CONWCOM, the first line of our conversation with the computer segment.

Meanwhile, we still have the exceptions for room 23. The problem we had here was that there was no light in this room, and without his flashlight our robot would be lost in the dark. This is one way of handling dark/light in a room. In line 50240 we check to see if that flashlight is off (L=0) or simply not a part of the robot's inventory (CO(8)<>1). If either of these is true, we print the message that it's too dark to see in the room. Then, if

CO(8) is simply not at hand, we GOTO 3000 for more conversation—the player is in the room, but he doesn't get to see anything! However, if the robot does have the flashlight, we allow him to turn it on of his own volition, and then we branch back to where we can begin displaying the description of the room. This leaves only one unconnected thread in our room layout section, the dangling command ELSE.

IF . . . THEN . . . ELSE. That is the question. IF . . . THEN hardly requires any description. IF something THEN something—it's perfectly obvious. But ELSE is a little more subtle, an extra shading that can do a lot for us but can also get a bit confusing if we're not careful with it. In its simplest form, which unfortunately is its least useful form, IF . . . THEN . . . ELSE is as obvious as a plain old IF . . . THEN. IF the condition is met THEN do something, otherwise (ELSE) do something else.

```
IF X=1776 THEN PRINT "THOMAS JEFFERSON" ELSE PRINT
"GEORGE III"
```

IF the condition of X=1776 is met THEN we do one thing, otherwise (ELSE) we do the other. Thinking of ELSE as "otherwise" should help keep it straight for you. Here's the first spin on the ball:

```
IF X=1776 THEN PRINT "THOMAS JEFFERSON":PRINT "VIR-
GINIA HAM"
```

The way your computer looks at an IF . . . THEN statement is this. IF the condition is met, processing continues along that same line. IF the condition is not met and there are no ELSEs in the line, processing drops down to the next line. So here, if X=1776, the computer will print both THOMAS JEFFERSON and VIRGINIA HAM. IF X<>1776, then processing will continue to the next line. It will not print 'VIRGINIA HAM.'

```
IF X=1776 THEN PRINT "THOMAS JEFFERSON" ELSE IF X=1783
PRINT "GEORGE WASHINGTON"
```

The computer looks at an IF . . . THEN . . . ELSE statement a little more complexly. IF the condition is met, that's easy enough, but if the condition is not met, then the computer scans the line looking for an ELSE, and then it measures that condition. In this example, if X=1776, then it will print THOMAS JEFFERSON. If X=1783, it will print GEORGE WASHINGTON. If X=something else, processing will continue on the next line. This leads us to

```
IF X=1776 THEN PRINT "THOMAS JEFFERSON" ELSE PRINT
"AARON BURR": PRINT "DAVID NIVEN"
```

Again, the computer checks to see if the first condition is met. If it is not, it scans the line for an ELSE and then acts accordingly. In this case, if X=1776, the computer will print THOMAS JEFFERSON but will *not* print DAVID NIVEN. If the first condition is met the computer does not then skip the ELSE like a skimming stone and jump to whatever is on the other side of the colon. It will print DAVID NIVEN only if X<>1776. And finally,

```
IF X=1776 THEN PRINT "THOMAS JEFFERSON" ELSE IF X=1860
PRINT "JEFFERSON DAVIS" ELSE IF X=1932 THEN PRINT
"BETTE DAVIS":PRINT "JEZEBEL"
```

Here we scan like crazy. If X=1776, then we print THOMAS JEFFERSON and go on to the next line; if X=1860, we print JEFFERSON DAVIS and go on to the next line; if X= 1932, we print BETTE DAVIS and JEZEBEL both; if X = none of those three values, processing simply passes to the next line without printing anything.

Running a few tests of your own with IF . . . THEN . . . ELSE can save you a lot of programming space, which is great, but used incorrectly they can cause you a lot of headaches, which is not so great. Remember, the ELSE is entirely optional; IF . . . THEN works perfectly well without it.

That takes care of the room display part of CONWCOM. So far we know that the main conversation with the computer begins at 3000 and that the room displays begin at 50000 (or 49900 if we're coming into it from INTRO). We also know that the routine that displays the objects lying around in each room is located at 13000. We have only a few more pieces to go before we've got all of CONWCOM scoped out in its entirety.

The next thing we should mention is the routine to save a game in progress. In fact, we've already discussed this in detail, and we've displayed it on page 50. This is the routine that begins at 30000, where all the pertinent game variables are written into a disk file called SDFILE. Looking back at it you'll see that the routine ends with a branch to 3000, the beginning of the conversation routine. Saving a game in progress does not mean ending that game; most adventurers make a point of saving a game every time they attempt a new bit of derring-do, just in case they get zapped in the process. After you've saved a game, the routine tells you that the game has in fact truly been saved (line 30130) and then sends you back to the beginning of the conversation segment at 3000, continuing exactly where you left off.

The last major part of CONWCOM is the routine that allows you to obtain an inventory. This programming, starting at 13100, is almost identical to the subroutine at 13000 that evaluates which movable objects are in a given room.

"Space Derelict!"

```
13100 FOR X = 1 TO 16
13110 IF CO(X) = 1 THEN FLAG = 1
    : X = 16
13120 NEXT
13130 IF FLAG = 1 THEN 13180
13140 PRINT "I am not carrying anything."
13150 GOTO 3000
13160 PRINT "I am carrying the following:"

13170 FOR X = 1 TO 16
13180 IF CO(X) = 1 THEN PRINT CO$(X);
13190 NEXT
13200 PRINT
13210 GOTO 3000
```

The only real difference between this and the room inventory is that here we're measuring the CO(X)s to see if they equal 1, while there we were looking for a CO(X) equal to 100+R. Also, 13000 was a subroutine with a RETURN, while 13100 is a plain routine with a GOTO to 3000 at the end. Otherwise it's practically line for line the same business in the same order.

Now, this next thing is a very small point, and there are other ways of doing this, but they are too complicated for the likes of me, or just plain unnecessary, so I'll tell you how I did it and either you can do it my way or else you can enter an assembly language routine or somesuch and do it your own way. I'm talking about line breaks. When you have a display like the one in line 13180, you have to account somehow for the fact that your words might not break precisely at the end of a line, so when your adventure displays it, it might look something like this:
I am carrying the following:
Small lead pentagon. Odd pie
ce of metal.
Some adventurers find themselves getting hungry enough to look around for the odd pie lying about, but it does somewhat defeat the professionalism we're trying for to have such sloppy leftovers lying on the counter.

As I say, you could probably come up with a little assembly language routine that would keep lines from breaking in midword, and you could probably even do it in Basic (perish the thought), but there's an easier way. Remember when I said to type in the list of movable objects *exactly* the way it was printed? If you did this, you discovered that I had typed in an awful lot of blank spaces that seemed unnecessary. But in fact they were quite necessary. You see, every CO$(X) string is exactly 20 characters long (with the exception of the SMALL AREA OF WATER and the FIVE-LEGGED ROBOT, neither of which is actually movable). And as you know, the length of the screen output is 80 characters long. So when this information in 13180 finally gets displayed, no matter how long it turns out to be, it is always divisible by 20 and therefore always breaks at a line ending. This may be the coward's way out of this predicament, but it works like a charm and it's highly recommended for my fellow cowards.

There are three other CONWCOM lines not actually part of the conversation that you would use in all your adventures. All three are rather straightforward.

"Space Derelict!"

```
16000 CHAIN"LOSER"
17000 CHAIN"WINNER"
63000 FOR PAUSE = 1 TO 2500
    : NEXT
    : RETURN
```

16000 is where we CHAIN to LOSER, and 17000 is where we CHAIN to WINNER. Note that in these situations we do not carry variable values. If we've won the game, it's over, and if we've lost, it's just as over. We're either finished entirely or starting again over from scratch; in either case, the variable values are of no use to us. And 63000, of course, is the pause we use occasionally within almost all our programs.

So altogether CONWCOM looks something like this:
3000 Conversation with the computer
13000 Room inventory
13100 Puppet inventory

```
16000 CHAIN "LOSER"
17000 CHAIN "WINNER"
30000 Save game
49900 COMMON game variables
50000 Open appropriate file and get room display
63000 The pause that refreshes
```

With only the most minor changes, all of this programming, with the exception of the conversation segment, could be carried intact from one adventure to the next. The only thing you might have to change is the number of COs measured in the 13000/13100 inventories and the 30000 save, and those odd room display exceptions that might occur at 50000. Otherwise it's programming for all seasons, a nice commodity that can save a lot of time on those cold winter nights in front of the computer.

And now on to the main segment of the conversation with the computer.

WORDS AND RESPONSES

Anyone who plays a few adventures by different designers soon finds that the commands that work for one don't always work for another. One game will allow you to GO NORTH and to GO DOOR, while another will accept any directional command but will spit the more specific GOs right back at you, requesting instead a direction. Some games will accept INV and INVENTORY and TAKE INVENTORY as well as such typing-creative variations as INVETNORY and TAKE INVNET (the program management of these niceties is simple and you'll see some of it in "Space Derelict!"). As a rule, the more adaptable your game is, the more fun for the player, but you can run yourself ragged trying to accommodate all the typing errors that in many cases ought simply to be shipped back to the player. You'll have to make up your own mind how adaptable you want to be, because it's your time that is going to be spent hacking at the computer.

We're going to list a bunch of words here, and while this list is in no way inclusive of all the vocabulary you can use in a game, it is a good shopping list to start with when you program your own conversation segments. Not all of these words are included in "Space Derelict!" by any means; those that are are marked by an asterisk. Also, not all of these words demand full programming space in the segment, as some are simply synonyms for others. TAKE and GET, for instance, are interchangeable, and the required actions are accounted for only at one stand in the program; if the other word is input by the player, the program simply substitutes the word it understands and acts accordingly (we'll see how that's done later).

The words below are grouped by length, as they are in the actual program.

One-letter:

N*
S*
E*
W*
U
D

(All of the above are, of course, directions.)

Two-letter:

GO*

Three-letter

RUN*
GET*
SAY*
EAT
BUY
FLY
SIT
TRY
CUT*
CRY*
TIE*

Four-letter:

READ*
TAKE*
SAVE*
QUIT*
SWIM
HIDE
GIVE
SING
KISS
PLAY
KICK*
HELP*
KILL*
OPEN*
LOOK*
DROP*
FIND*
MELT*
LOCK*
MAKE
PUSH*

Five-letter:

CLIMB*
POINT
LIGHT*
BUILD
LEAVE*
SCORE*

SHOOT*
START
DRINK
BRIBE
WHERE*
SMOKE
FRISK
PRESS*
ENTER*
CLICK*
TOUCH*
THROW*
LASSO*
BLAST*
THINK*
BREAK
SMELL

Six-letter:

UNLOCK*
TICKLE
SEARCH
INSERT*
REMOVE*

Seven-letter:

WRESTLE
UNLIGHT*
EXAMINE*

Eight-letter:

DESCRIBE*

Nine-letter:

INVENTORY*

As you can see, we've got quite a range here, and even as you read them you probably thought of a dozen more that would have fit in nicely. Some of these are quite specific to precise circumstances in a given game universe, but others are general enough to merit inclusion in every adventure. This list is merely a starting point for your own creativity; don't take it as gospel because it's only one man's list of words I like, and you can amend, adjust, and expand it as you see fit.

The responses the computer displays on the video screen must be specific to your game and the situation in which the player finds himself at the moment. In "Space Derelict!" since these responses are coming from a robot there is a certain deliberate robotic woodenness to the wording that wouldn't work for other games. You might want to go back now to the listings from 2030 to 2250, which we covered in the previous chapter, to follow the way the responses were handled in ''Space Derelict!'' You'll recall that our catchall array for answers from the computer was the string variable A$(X).

A$(1) answers the situation mentioned earlier in this chapter when the input is somewhat recognizable but the combination of the words doesn't make sense in the program. Let's say the player commands CLIMB HOLOGRAM, for instance. The program will recognize CLIMB, but nothing in the CLIMB section will account for going up the hologram, so the program responds in such a way as not to discourage the player from climbing but to tell him that at the moment he's up the wrong tree. A$(2) is the other half of the coin, where the first word of the command is not recognized by the program. When that word isn't found, the program prints A$(2) *plus* the word in question (although that isn't obvious just by looking at line 2040—please stay tuned for specifics), and that way the player knows not to try that word again. Also, if the player has input a typo, this way he will see delivered back to him exactly what he typed in, so he'll know the mistake was mechanical and not metaphysical.

A$(3) is so optional that I don't think it's ever used in ''Space Derelict!'' even though it's set so nicely right at the beginning. I believe that every time the computer accepts a command it should have a default response of OKAY, but it's a hell of a lot easier to type OKAY than it is A$(3) all the time. Suit yourself with this one; you'll find ''Space Derelict!'' loaded with OKAYs, in any case.

A$(4) and A$(5) are self-explanatory. A$(4) comes up when a player tries to go north or wherever and there's no north to go to, while A$(5) reminds him that the door ahead is locked.

A$(6) takes care of situations where the player is trying to do something it's okay to do but not at the moment. CLIMB ROPE, for instance, is perfectly acceptable in ''Space Derelict!'' as there is a rope in room 22 just ripe for the climbing. But if our player is in room 4, where there is no rope, we have to handle CLIMB ROPE in such a way as merely to postpone it, not to cancel it entirely. A$(6) does this for us.

A$(7) is the response to commands to somehow manipulate an object not presently available and is much like A$(6) in that it keeps the player hopeful that whatever he's looking for will soon turn up.

A$(8) answers those commands where the player is trying to get the program to solve the problem for him. For instance, let's say the player comes upon the first locked door in room 3 and simply commands UNLOCK DOOR. Of course he wants to unlock the door, but he's got to discover the means of so doing first, hence the response A$(8), which puts him in his place and lets him know that he's going to have to do all the work himself.

A$(9) is much like A$(1) but is a little more precise and leading. We'll see how it's used when we get further into the program.

A$(10) through A$(20) are descriptions of various items that will be supplied if the player requests them. They are loaded here rather than in the conversation section as much to save space on the programming lines where they occur as to save program space per se. Any one programming line can contain only 240 characters, and there is a danger with some of those lines of exceeding that limitation.

A$(21) is the response to commands that just don't work, like trying to click the door opener at a door that's already open, or at a door that opens to a different command.

A$(22) and A$(23) are two more descriptions, A$(22) being of the non-descriptive variety, the response to describe something that doesn't require more description, and A$(23) obviously being the robot's laser.

There are other responses from the program buried away in the conversation segment, occurring at those one-of-a-kind situations where it was easier to include it there than to come back and amend the A$(X) variables. But you can see here a general pattern of using variables for the more frequently necessary responses—no can do, nothing happens, wrong direction, etc.—that you should include in your own adventures to save yourself programming time when you dive into the actual conversation segment.

THE MECHANICS OF THE CONVERSATION SEGMENT

The first step in the conversation segment is to break down the player input into a form recognizable to the program. Since the segment is arranged according to the length of the first word of the input (or only word, if it's a one-word input), the first thing we do is get the length of that first word. We might as well go now to the program to see how it's done.

"Space Derelict!"

```
3000  REM  CONWCOM
3010  IF R = 23 THEN GOSUB 40000
3020  PRINT
3030  PRINT CHR$ (14)
3040  INPUT "-->What should I do";C$
3050  PRINT CHR$ (15)
3060  IF C$ = "" THEN PRINT A$(1)
    : GOTO 3000
3070  IF LEN (C$) = 1 THEN 3180
3080  IF MID$ (C$,2,1) = " " THEN PRINT A$
        (2) + LEFT$ (C$,1)
    : GOTO 3000
3090  IF LEN (C$) = 2 OR MID$ (C$,3,1) = "
        " THEN 3840
3100  IF LEN (C$) = 3 OR MID$ (C$,4,1) = "
        " THEN 4390
3110  IF LEN (C$) = 4 OR MID$ (C$,5,1) = "
        " THEN 4790
3120  IF LEN (C$) = 5 OR MID$ (C$,6,1) = "
        " THEN 6850
3130  IF LEN (C$) = 6 OR MID$ (C$,7,1) = "
        " THEN 7570
3140  IF LEN (C$) = 7 OR MID$ (C$,8,1) = "
        " THEN 7790
```

```
3150  IF LEN (C$) = 8 OR MID$ (C$,9,1) = "
         " THEN 7870
3160  IF LEN (C$) = 9 OR MID$ (C$,10,1) =
         " " THEN 7890
3170  PRINT A$(2) + C$
     : GOTO 3000
```

Line 3000 reminds us where in the program we are, and line 3010 is an exception to all adventure-design rules, a piece of miscellany that doesn't fit anywhere else (we'll describe miscellany in detail at the end of this chapter). As you can see, this line comes into play only if the player is in room 23, so it certainly doesn't affect most of the game and can easily be ignored for the time being.

Skipping line 3030 momentarily, line 3040 gets the player input, which will be the string C$. You can rephrase your question in this line any way you want if you're not quite satisfied with—>What should I do? but keep in mind that as programmed the INPUT statement will print the player's input on the same line as these words as it is typed in, so your prompt words should be kept short for the player's input to fit all on one line.

Following the collection of the input, line 3050 turns off the cursor by printing a Control O. This way we don't see the blinking light while other processing—like getting a room/display—is being performed. This is just a little nicety that makes what's on the screen that much more attractive. Line 3030 turns it on again before the next prompt for a player input.

Lines 3060 through 3160 measure the length of the first word of the input, then route the program to the appropriate area to handle words of that length. All possible lengths, including null strings of zero length, are accounted for. Line 3060 is the null-string handler; the only time a string will come in null is when the player hits the Enter button before he puts in his C$, so A$(1) simply tells him to get his act together, and then the program routes back to 3000. Keep in mind that not only does the player see A$(1) printed on the screen, but also the—>What should I do? prompt again as well printed below it.

Line 3070 accounts for one-letter-long inputs, routing the program to line 3180, the beginning of the one-letter group. Line 3080 is in effect a typo handler. It reads C$ and, if C$ is longer than one letter (the program would have already branched from 3070 if it was one letter), this line looks for a space as the second character. We're looking for input here along the lines of G DEVICE or I ICE—meaningless stuff that needs to be sent back to the player, with the A$(2) explanation concatenated to the screwy one-letter first-word input that caused the problem, in this case LEFT$(C$,1). If in fact the input had been G DEVICE, then the player would have seen:
I'm sorry, but I cannot respond to the command G
—>What should I do?
A$(2) is designed to tell the player what it can't respond to. You'll notice that a similar error catchall does not follow the subsequent length-checking lines. Rest assured that this function is carried out within the programming for groups of these lengths.

Lines 3090 through 3160 are virtually identical, so we'll analyze just one of them. The first thing 3090 does is figure if C$ is two characters long. If that is the case, processing will continue along this line—i.e., a branch to

3840. The OR provision figures whether the third character of C$ is a space, which would mean, of course, that the first word of C$ is two letters long. Now we have accounted for either a one-word input two characters long or an input of any length where the first space found is the third character, meaning again that the first word is two characters long. Keep in mind that each successive line of the program moves the measurement of that space character one character along: Line 3080 looks for the space at character number 2, 3090 looks for it at character number 3, and so on, so if that space criterion is not met, the processing simply continues one line further down. Each of these lines from 3090 through 3160 provides a branch to the appropriate word-length group for lengths from two to nine characters. Line 3170 is our default, accounting for first word-length inputs of 10 characters or more; once again, A$(2) plus this time all of C$, whatever its length, are returned to the player as unacceptable. And we've now accounted for every possible length and variation of C$. We haven't done much with it yet, but at least we know what it is and, if we're going to act upon it, how long it is. This length information will remain important as we continue processing the data contained in C$.

MOVING AROUND

There's a lot of programming in the conversation-with-the-computer segment, and much of it is repetitious. While we will go through all of it in this chapter, there is really no point in explaining every single line when we've covered the gist of that line earlier. Take moving from room to room, for instance. The programming lines covering this extend from 3180 through 3830, chock full of the same things over and over and over again. So there will be a little less detail as we progress through this section, but we will cover everything as clearly as possible.

Movement from room to room will stem from the player inputting either a direction such as N or E or the more complete input GO NORTH or GO EAST. The section of the program covering this movement is broken down into each of the four possible directions, so if the player inputs W then the computer processes the west section to act upon that input. Two things need to be figured out by the program in order to act. First, it must determine whether the puppet can in fact move in that direction. If the player wants to go west but there is no west exit, the program must come back to the player with A$(4), "I cannot go in that direction," and then reroute back to line 3000. Second, if there *is* a west exit, then the program must determine whether the door between the player and the room to the west is open or shut. This is done by checking the value of the appropriate D(X). If the door is shut, then the program must return to the player with A$(5), "There is a locked door in that direction," and a reroute back to 3000. If it's open, then the program must move the player into that west room. The actual movement is handled very simply through changing the value of the variable R. If the player is in room 4, for instance, and he enters the GO WEST command, the program reassigns a value of 3 to the variable R and then routes through the room-setting routine at 50000, finally coming back to 3000 from the direction at the end of the 50000 routine. All very simple, but the program must evaluate all these variations for each and every possible room.

Sitting down to program this section is not hard, just time-consuming. Start with any direction (''Space Derelict!'' starts with north), then run through the three procedures for each room. The three procedures again are: is/isn't exit in that direction; door locked/not locked; stay put and back to 3000/make move then back to 3000. Let's look at these lines in ''Space Derelict!'' and cover the important points.

''Space Derelict!''

```
3180  IF C$ = "S" THEN 3470
3190  IF C$ = "N" THEN 3230
3200  IF C$ = "W" THEN 3620
3210  IF C$ = "E" THEN 3730
3220  PRINT A$(1)
    : GOTO 3000
3230  IF R < 3 THEN 3220
3240  IF N < > 1 THEN PRINT A$(4)
    : GOTO 3000
3250  IF R = 5 THEN R = 11
    : PRINT "Okay."
    : GOTO 50000
3260  IF R = 6 AND D(1) = 0 THEN PRINT A$(
        5)
    : GOTO 3000
3270  IF R = 6 AND D(1) = 1 THEN R = 8
    : PRINT "Okay."
    : GOTO 50000
3280  IF R = 7 AND D(2) = 1 THEN R = 6
    : PRINT "Okay."
    : GOTO 50000
3290  IF R = 7 AND D(2) = 0 THEN PRINT A$(
        5)
    : GOTO 3000
3300  IF R = 8 THEN PRINT "Okay."
    : R = 9
    : GOTO 50000
3310  IF R = 10 AND D(4) = 1 THEN R = 22
    : PRINT "Okay."
    : GOTO 50000
3320  IF R = 10 THEN PRINT A$(5)
    : GOTO 3000
3330  IF R = 11 THEN R = 4
    : PRINT "Okay."
    : GOTO 50000
3340  IF R = 12 AND D(3) = 1 THEN PRINT "O
        kay."
    : R = 10
    : GOTO 50000
3350  IF R = 12 THEN PRINT A$(5)
    : GOTO 3000
3360  IF R = 18 AND D(5) = 1 THEN R = 21
    : PRINT "Okay."
    : GOTO 50000
```

```
3370  IF R = 18 THEN PRINT A$(5)
    : GOTO 3000
3380  IF R = 19 AND D(6) = 1 THEN R = 18
    : PRINT "Okay."
    : GOTO 50000
3390  IF R = 19 THEN PRINT A$(5)
    : GOTO 3000
3400  IF R = 21 AND D(7) = 1 THEN R = 23
    : PRINT "Okay."
    : GOTO 50000
3410  IF R = 21 THEN PRINT A$(5)
    : GOTO 3000
3420  IF R = 22 AND D(8) = 0 THEN PRINT A$
        (5)
    : GOTO 3000
3430  IF R = 22 THEN PRINT "Okay."
    : R = 23
    : GOTO 50000
3440  IF R = 23 AND D(9) = 1 THEN R = 24
    : PRINT "Okay."
    : GOTO 50000
3450  IF R = 23 THEN PRINT A$(5)
    : GOTO 3000
3460  GOTO 3220
3470  IF R < 3 THEN PRINT A$(1)
    : GOTO 3000
3480  IF S < > 1 THEN PRINT A$(4)
    : GOTO 3000
3490  IF R = 3 AND D(10) = 1 THEN R = 2
    : PRINT "Okay."
    : GOTO 50000
3500  IF R = 4 THEN R = 11
    : PRINT "Okay."
    : GOTO 50000
3510  IF R = 11 AND D(11) = 1 THEN R = 5
    : PRINT "Okay."
    : GOTO 50000
3520  IF R = 11 THEN 16000
3530  IF R = 6 AND D(2) = 1 THEN R = 7
    : PRINT "Okay."
    : GOTO 50000
3540  IF R = 8 AND D(1) = 1 THEN R = 6
    : PRINT "Okay."
    : GOTO 50000
3550  IF R = 9 THEN R = 8
    : PRINT "Okay."
    : GOTO 50000
3560  IF R = 10 AND D(3) = 1 THEN PRINT "O
        kay."
    : R = 12
    : GOTO 50000
3570  IF R = 18 AND D(6) = 1 THEN PRINT "O
        kay."
    : R = 19
    : GOTO 50000
```

```
3580  IF R = 21 AND D(5) = 1 THEN PRINT "O
        kay."
     : R = 18
     : GOTO 50000
3590  IF R = 22 AND D(4) = 1 THEN R = 10
     : PRINT "Okay."
     : GOTO 50000
3600  IF R = 24 THEN R = 23
     : PRINT "Okay."
     : GOTO 50000
3610  PRINT A$(5)
     : GOTO 3000
3620  IF R < 3 THEN PRINT A$(1)
     : GOTO 3000
3630  IF W < > 1 THEN PRINT A$(4)
     : GOTO 3000
3640  IF R = 3 AND D(12) = 1 THEN R = 6
     : PRINT "Okay."
     : GOTO 50000
3650  IF R = 4 THEN R = 3
     : PRINT "Okay."
     : GOTO 50000
3660  IF R = 9 THEN R = 10
     : PRINT "Okay."
     : GOTO 50000
3670  IF R = 14 AND D(14) = 1 THEN R = 15
     : PRINT "Okay."
     : GOTO 50000
3680  IF R = 16 THEN R = 9
     : PRINT "Okay."
     : GOTO 50000
3690  IF R = 17 AND D(17) = 1 THEN R = 15
     : PRINT "Okay."
     : GOTO 50000
3700  IF R = 18 THEN R = 17
     : PRINT "Okay."
     : GOTO 50000
3710  IF R = 20 THEN R = 18
     : PRINT "Okay."
     : GOTO 50000
3720  PRINT A$(5)
     : GOTO 3000
3730  IF R < 3 THEN PRINT A$(1)
     : GOTO 3000
3740  IF E < > 1 THEN PRINT A$(4)
     : GOTO 3000
3750  IF R = 3 THEN R = 4
     : PRINT "Okay."
     : GOTO 50000
3760  IF R = 6 AND D(12) = 1 THEN R = 3
     : PRINT "Okay."
     : GOTO 50000
3770  IF R = 9 THEN R = 16
     : PRINT "Okay."
     : GOTO 50000
3780  IF R = 10 THEN R = 9
     : PRINT "Okay."
     : GOTO 50000
```

```
3790  IF R = 13 AND D(15) = 1 THEN PRINT "
        Okay."
    : R = 15
    : GOTO 50000
3800  IF R = 16 AND D(16) = 1 THEN R = 15
    : PRINT "Okay."
    : GOTO 50000
3810  IF R = 17 THEN R = 18
    : PRINT "Okay."
    : GOTO 50000
3820  IF R = 18 THEN R = 20
    : PRINT "Okay."
    : GOTO 50000
3830  PRINT A$(5)
    : GOTO 3000
```

Lines 3180 through 3210 evaluate the contents of a one-letter C$ input, branching to the appropriate direction in the rest of this section. S, N, W, and E are accounted for with branches, while any other one-letter input gets the A$(1) raspberry and an express train back to 3000.

So let's imagine we're programming now, and we're going to evaluate all the rooms in response to the N command. We'll be starting here at 3230. First we take rooms 1 and 2, which happen to share a similar situation. If you look at the descriptions of these rooms in the RMAKER routine, you'll see that no exit information is given. The only way the player can proceed from room 1 to room 2 is with an ENTER AIRLOCK or GO AIRLOCK. The same is true for room 2, where ENTER SHIP or GO SHIP must be used, so if R<3 then we return the response of A$(1), unacceptable instruction, which is more germane than the specific A$(4), can't-go-in-that-direction. Even though in these rooms N (our north value set in RMAKER) does not equal 1, it's fairer to the player to give him A$(1) over A$(4). For any other room where N does not equal 1, however, we do give our player the A$(4), in line 3240, with a reroute back to 3000. With this line we dismiss all attempts to go north in rooms where going north is an impossibility. Similar programming handles the other directions later on.

So we move along to room 5, the first one that does have a north exit and no door, so when a player is in room 5 he can unconditionally go north. Looking at the map, we see that north of room 5 is room 11; hence line 3250, which changes the value of R and then branches to 50000. You'll notice too the response "OKAY", just because it's polite to answer people when they talk to you. You'll also notice that this proves the author's reluctance to type in A$(3) when a simple "Okay" will suffice.

Room 6 gives us our final possibility, as there is a door to the north of this room (which happens to be D(1)). If D(1)=0, which means a closed door, then we print A$(5), advising our player of the situation and branching back to 3000. This is done in 3260. Line 3270 handles the other possibility and moves the player to room 8 accordingly. Note that line 3270 need not include the AND D(1) = 1, because if it doesn't equal 0 than it *has* to equal 1. This concept of logical exclusion is simple: If there are only two ways of doing something and you've just eliminated the first one, then you have only the second one left to contend with. Lines 3310–3320 show you logical exclusion at work in actual programming statements. Line 3310 covers the possibility of our being in room 10 with the door open. The only other possible door condition if we are in room 10 has to be door closed, and the only way line 3320 will be processed if R does equal 10 is if D(4) is equal to 0. Therefore all we have to write in 3320 is IF R = 10, since the door condition is implicit in our having gotten to this line in the first place. As you can see, logical exclusion certainly can help streamline our programming, but it suffers at times from being somewhat unclear to the person coming at our program with less than our complete understanding of it. For the sake of clarity I've kept down the use of logical exclusions in many cases where the program could be polished and tight. In the poker program we will use it all the time, so we'll ease into it now in "Space Derelict!"

The rest of the lines through 3450 cover all the rooms going north, doing exactly the same things we've just been doing, so I won't go into them any further. Line 3460 is a simple redundancy that can easily be eliminated, but I've ended each of the four directional segments with a similar line just to be on the safe side. It doesn't hurt, and it helps me sleep better at night just knowing it's there. If there were a logical error in one of these segments these catches would make that unequivocally apparent at the debugging stage, which makes for easier debugging than leaving it all to chance.

The rest of the lines up through 3830 simply recapitulate the ones above, but for the other three directions, and with one exception there's simply no point in going into them again here; we've got programming of a lot more interesting nature ahead of us. But there is one variation that needs to be pointed out. You'll remember that room 5 is a trap and that entering room 5 without first "opening" the imaginary door that deactivates the lethal device in the ceiling will kill the puppet robot. So we account for this in lines 3510 and 3520. If the door is open, hunky-dory, and we enter the room as usual. But if D(11)=0 then we branch to line 16000, the CHAIN to the losing routine. The player does get to look longingly into room 5 from room 11, but he'll never get in room 5 until he deactivates the deadly device. And another one bites the dust.

GOING OTHER PLACES—TWO-LETTER WORDS

You have a decision to make at the outset of your designing: Is your game only going to respond to directional commands, or can the player get to a specific place or item by such commands as GO DOOR and GO TABLE? I have seen games designed both ways, and quite frankly I prefer the latter as a player. When you see something sitting right in front of you in a room—a chair, for instance—it is mightily frustrating to type in GO CHAIR only to get the response "I need a direction." Not that it's so hard to map this sort of thing as a player, but it is a pain in the neck. It is a tad harder on the programming side, obviously, because not only do you need to travel from room to room in response to directional commands, but also from place to place within rooms. If you prefer the directional-command approach, all you have to do is adjust the section we've just gone through with an eye on some of the things we'll be discussing in this section.

We're going to introduce a new twist to our analysis of C$ this time out. We already figured out the length of the first word back at the outset of the

conversation-with-the-computer routine. Now we're going to refine that one step further by giving a name and a definition to all the rest of C$. We know that if we have branched to line 3840 we are dealing with a C$ that consists of (1) a two-letter word followed by (2) a space. What follows that space is still a mystery, but at least if we quantify it we can hold it up to the light a little better. So we're going to enter the following line (plus lines similar to it for each consecutive word-length grouping):
S2$ = MID$ (C$,3)
This gives us the new string S2$, which consists of all of C$ *starting with the space*. This distinction is very important both for typographical reasons (we'll always have to include the space in our programming lines, as you'll soon see) and for LEN length reasons, due to various logical exclusions we will encounter (usually at the beginning of length groupings). You'll see what I'm talking about as we progress; the important thing is that we now have something other than the first word of C$ to analyze. And analyze it we will. Keep in mind that the use of MID$ here takes C$ from character number 3 all the way to the end of the string, regardless of its length. The reason that we don't define S2$ as C$(MID$,4)—i.e., starting at the first letter after the space—is that C$ could have a LEN of only 3 as defined in line 3090, and therefore C$(MID$,4) would give us a useless null string.

We should go now to the relevant programming lines in "Space Derelict!" for a more specific discussion.

"Space Derelict!"

```
3840  REM  2 LETTER 1ST WORDS
3850  IF C$ = "GO" THEN PRINT A$(1)
    : GOTO 3000
3860  IF LEFT$ (C$,2) < > "GO" THEN
        PRINT A$(2) + C$
    : GOTO 3000
3870  S2$ = MID$ (C$,3)
3880  IF C$ = "GO NORTH" THEN 3230
3890  IF C$ = "GO SOUTH" THEN 3470
3900  IF C$ = "GO WEST" THEN 3620
3910  IF C$ = "GO EAST" THEN 3730
3920  IF R < > 23 THEN 3970
3930  IF C$ = "GO SOUTHWEST" AND D(8) = 1
        THEN PRINT "Okay."
    : R = 22
    : GOTO 50000
3940  IF C$ = "GO SOUTHWEST" THEN PRINT A$
        (5)
    : GOTO 3000
3950  IF C$ = "GO SOUTHEAST" AND D(7) = 1
        THEN R = 21
    : PRINT "Okay."
    : GOTO 50000
3960  IF C$ = "GO SOUTHEAST" THEN PRINT A$
        (5)
    : GOTO 3000
3970  IF R < > 15 THEN 4060
```

```
3980  IF C$ = "GO NORTHEAST" AND D(17) = 1
        THEN R = 17
     : PRINT "Okay."
     : GOTO 50000
3990  IF C$ = "GO SOUTHEAST" AND D(14) = 1
        THEN R = 14
     : PRINT "Okay."
     : GOTO 50000
4000  IF C$ = "GO NORTHWEST" AND D(16) = 1
        THEN R = 16
     : PRINT "Okay."
     : GOTO 50000
4010  IF C$ = "GO SOUTHWEST" AND D(15) = 1
        THEN R = 13
     : PRINT "Okay."
     : GOTO 50000
4020  IF S2$ = "GO SOUTHWEST" OR S2$ = "GO
         NORTHWEST" OR S2$ = "GO SOUTHEAST
       " OR S2$ = "GO NORTHEAST" THEN
        PRINT A$(5)
     : GOTO 3000
4030  IF C$ = "GO PANEL" THEN PRINT "There
         are numerous controls here, but n
        one of them seem to be operative."

     : GOTO 3000
4040  IF C$ = "GO PEDESTAL" AND CO(6) <
        > 0 AND CO(6) < > 115 THEN PRINT "
        Okay."
     : GOTO 3000
4050  IF C$ = "GO PEDESTAL" THEN CO(6) = 1
        15
     : PRINT "There's something on top of t
        he pedestal"
     : GOSUB 13000
     : GOTO 3000
4060  IF C$ = "GO HOLOGRAM" AND R = 4
        THEN PRINT "It appears to be some
        sort of detailed map, but the char
        acters and layout are  incomprehe
        nsible."
     : IF CO(3) = 1 THEN PRINT "I am receiv
        ing a strange subverbal message. I
        t says 'Think, Earthling.'"
4070  IF C$ = "GO HOLOGRAM" AND R = 4
        THEN 3000
4080  IF S2$ = " HOLOGRAM" AND R < > 4
        THEN PRINT A$(7)
     : GOTO 3000
4090  IF S2$ = " PANEL" AND (R = 2 OR R =
        7 OR R = 24) THEN PRINT "Okay."
     : GOTO 3000
4100  IF S2$ = " PANEL" THEN PRINT A$(7)
     : GOTO 3000
4110  IF S2$ = " AIRLOCK" AND R = 1 THEN
        PRINT "Okay."
     : R = 2
     : GOTO 50000
```

```
4120  IF S2$ = " AIRLOCK" THEN PRINT A$(1)
     : GOTO 3000
4130  IF S2$ = " DOOR" THEN PRINT "Okay."
     : GOTO 3000
4140  IF S2$ = " SHIP" AND R = 1 THEN R =
        2
     : GOTO 50000
4150  IF S2$ = " SHIP" AND R = 2 AND D(10)
        = 1 THEN R = 3
     : GOTO 50000
4160  IF S2$ = " SHIP" AND R = 2 THEN
        PRINT A$(5)
     : GOTO 3000
4170  IF S2$ = " SHIP" THEN PRINT A$(1)
     : GOTO 3000
4180  IF CO(3) = 107 AND LEFT$ (S2$,4) = "
         PED" AND R = 7 THEN PRINT "Okay.
        That is where the crystal is."
     : GOTO 3000
4190  IF LEFT$ (S2$,4) = " PED" AND R = 7
        THEN PRINT "Okay."
     : GOTO 3000
4200  IF LEFT$ (S2$,4) = " BED" THEN
        PRINT "Series R Robots have no nee
        d of sleep."
     : GOTO 3000
4210  IF LEFT$ (S2$,6) = " CHAIR" AND R =
        15 THEN PRINT "Okay."
     : PRINT A$(22)
     : GOTO 3000
4220  IF LEFT$ (S2$,6) = " CHAIR" THEN
        PRINT A$(7)
     : GOTO 3000
4230  IF S2$ = " TUBES" AND (R = 13 OR R
        = 14) THEN PRINT A$(22)
     : GOTO 3000
4240  IF S2$ = " TUBES" THEN PRINT A$(7)
     : GOTO 3000
4250  IF R = 21 AND ( LEFT$ (S2$,6) = " TA
        BLE" OR S2$ = " SCULLERY" OR
        LEFT$ (S2$,5) = " OVEN" OR S2$ = "
         CUPBOARD") THEN PRINT A$(22)
     : GOTO 3000
4260  IF R = 22 AND (S2$ = " COURT" OR
        LEFT$ (S2$,5) = " ROPE" OR S2$ = "
         POOL") THEN PRINT "Okay."
     : GOTO 3000
4270  IF S2$ = " COURT" OR LEFT$ (S2$,5)
        = " ROPE" OR S2$ = " POOL" THEN
        PRINT A$(7)
     : GOTO 3000
4280  IF C$ < > "GO DEVICE" THEN 4380
4290  IF R = 5 THEN PRINT "It is inaccessi
        ble."
     : GOTO 3000
4300  IF R < > 4 THEN PRINT A$(7)
     : GOTO 3000
```

```
4310  PRINT "It seems to be some sort of t
        ransmitting device. It appears to
        be operative.     Please stand by."

    : GOSUB 63000
4320  PRINT "It is sending a message in an
         electronic code. The main memory
        banks of the con-trol computer sho
        uld be able to translate it. I am
        switching control to main     circu
        its. Please stand by."
    : GOSUB 63000
    : GOSUB 63000
    : GOSUB 63000
4330  PRINT "TO THE PEOPLE OF THE SYSTEM O
        F THE NINE PLANETS: THIS IS A MESS
        AGE FROM THE      UNITED FEDERATION
         OF KRENN."
    : PRINT
4340  PRINT "THIS SHIP IS A MICRONUCLEAR E
        XPLOSIVE DEVICE. IT IS AIMED DIREC
        TLY AT YOUR STAR,AND WILL EXPLODE
        WHEN IT REACHES  A 100,000,000 MIL
        E RADIUS. THIS WILL PUT IT    ALMOS
        T DIRECTLY NEXT TO YOUR HOME PLANE
        T."
4350  PRINT
    : PRINT "THE REASON FOR THIS ACTION IS
         SIMPLE. NOW THAT YOU HAVE DEVELOP
        ED FASTER THAN    LIGHT TECHNOLOGY
        YOU ARE A THREAT TO THE GALACTIC D
        OMINANCE OF THE KRENN. THERE-FORE
        YOU MUST BE DESTROYED."
4360  PRINT
    : PRINT "FAREWELL, PEOPLE OF THE SYSTE
        M OF THE NINE PLANETS."
4370  PRINT
    : PRINT "END TRANSMISSION."
    : PRINT
    : GOTO 3000
4380  PRINT A$(9)
    : GOTO 3000
```

Line 3850 is one of a sort that will pop up frequently. Our program knows what GO means, but it needs more information to act upon it. This line delivers that message.

Since the entire two-letter-word section is comprised of nothing but GO, no other two-letter words are acceptable, hence line 3860. And 3870 quantifies S2$ as we just discussed above. Lines 3880–3910 introduce us to the concept of synonyms. If there are two or more ways of saying something, use them all, since all we have to do is reference the variations back to the first usage. Here we do that with GO NORTH referenced back to N, GO SOUTH back to S, and so forth. You'll see a lot more of this as we go along.

Lines 3930–3960 refer to situations encountered only in room 23, which explains line 3920, while 3970–4050 refer to situations found in room 15. Most of these lines refer to the somewhat confusing multidirectional doors I introduced into this room. They may not have been such a bad idea for the player, but they did require special programming space to handle them. Beware of such game elements that cause more difficulties than they may be worth. In any case, the same programming structure we applied to the N, S, E, and W section applies to these. Lines 4030–4050 introduce special GO situations like those we'll meet up with in the remainder of this section. There are both a panel and a pedestal in room 15. Line 4030 describes the panel if the player has decided to go to it. Lines 4040 and 4050 describe the pedestal, both with and without the crystal star, CO(6) .

The important thing to remember is that your player *must* be able to move *everywhere* there is to move. If you look at the on-screen description of room 15, for instance, you will see that it specifically mentions both the panel and pedestal. Even if these are inconsequential room decorations (as is the panel), it is imperative that the program nonetheless recognize their existence. It would be somewhat second-rate for the player to input GO PANEL, only to get a reply I DON'T KNOW WHAT A PANEL IS. It's there, so it has to be accounted for. This means a lot of somewhat tedious programming, but it also means that your game universe is complete, with all its components "real" insofar as both the player and the program are concerned. Line 4130 is the supreme example of this. Since there are doorways of one sort or another in every room, whenever a player says GO DOOR the program replies with "Okay," as if the robot has been moved up to the door. Nothing happens, as you can see, but at least the player thinks something happens, and that's the important thing.

You'll notice that there's no cowcatcher between 4050 and the rest of the GOs—i.e., there's no catchall barrier in the program to handle the fact that the condition of being in room 15 is no longer met. That's because all the rest of the conditions apply equally to room 15 and to the rest of the rooms.

Lines 4060–4070 handle the hologram in room 4. The IF CO(3) = 1 business is the clue to the use of the crystal pentagon. Note that if the CO(3) = 1 condition is not met, processing will not continue along this line through the colons and will instead go on to 4070.

Let us now digress for a minute. We've been dealing with something here all along without defining it because the operation has been so obvious as not to bear further discussion. But it's something we can't ignore any longer because we're going to start stretching it to its limits. That something is called Boolean logic.

When we say something like:

```
4080 IF S2$ = " HOLOGRAM" AND R< > 4 THEN PRINT A$(7):
GOTO 3000
```

the AND statement is a Boolean operator. The computer reads this statement and, provided both AND conditions are met, does what is asked of it regardless of what those conditions are. As far as the computer is concerned, we have spoken the truth, so the computer applies a TRUE to that situation (which, if you're really interested, is a value of – 1; FALSE would be a value of 0) and does what we ask of it. If either one of those conditions is not met, the computer does not do what we ask of it.

OR is another Boolean operator. In line 4090 we say:

```
4090 IF S2$ = " PANEL" AND (R = 2 OR R = 7 OR R = 24) THEN
PRINT "OKAY." : GOTO 3000
```

which incorporates both AND and OR. With OR all you have to do is get one TRUE for the computer to obey your commands. In line 4090 we compound our Booleania by first getting some truth out of what's in the parentheses and adding this truth to the truth of S2$. When the computer comes across this line it first looks inside the parentheses—parenthetical statements take precedence in the actual processing. For the computer to come up with a TRUE for the ORs all we need is one of these values of R to be accurate. If none of them is accurate, we get a FALSE. After handling the parentheses the computer goes back and measures the rest of the line. Now it's looking for truth on both sides of the AND. If S2$ does equal " PANEL" at this point, but R equals none of these values, nothing will happen because the FALSE overrides the TRUE in the AND situation. Two TRUEs equal truth. And as in the real world, one FALSE plus one TRUE do not equal the truth, and two FALSEs also do not equal the truth.

As I said, most of the time you handle OR and AND situations without the slightest concern for the Booleanism of the thing, but I mention it now because of the way it is compounded by the parentheses in 4090 (and also further along in both this and the poker program). Using this kind of seemingly straightforward logic can eventually lead to some pretty complicated-looking programming statements, so knowing what it's all about won't hurt. The interesting facet to all of this is that you can indeed use parentheses for nonmathematical statements. We're used to seeing things such as:

```
X = C/( (Y+2)*(Z-Y) )
```

but we can also say things such as:

```
IF ( (A$ = "WHEATIES" OR B$ = "SALAMI") AND (C$ = "MEA-
SLES" OR D$ = "PLAGUE") ) AND E$ = "TIME IS ON OUR SIDE"
THEN PRINT "WHAT???"
```

and have the computer make sense of it.

The last thing worth noting here is that the computer measures all parts of Boolean statements, even after the condition we're looking for is met. For example, let's say we have this line:

```
IF X = 5 OR (10/Y > 0) THEN Z = 11
```

and let's further assume at this point X does equal 5 and Y equals 0. First off, the computer measures X = 5 and finds truth. The problem is, despite having satisfied the conditional—if X = 5 it doesn't matter anymore whether 10/Y>0 because of the OR—the computer nonetheless goes ahead and measures 10/Y>0 anyhow. And what would happen here is a break in the program for a division by zero error. Beware of such potential disasters in your logic evaluations.

So much for Boolean logic for now.

The rest of this section in "Space Derelict!" pretty much does what we've been doing all along, measuring for the correct situations and either acting upon them if the criteria are fulfilled or sending them back to the player if they're not. Therefore I won't go into these lines any further, with two exceptions. First, there are some beds scattered around in "Space Derelict!"; hence line 4200. This is, in fact, a joke, and the more jokes you can put into your game, the better. But this line also makes another point.

You'll notice that here I have used the LEFT$ statement on BED. This is to allow the possibility that the player might have originally typed in GO BEDS as well as GO BED, given the game situation. This way, the program handles both possibilities. Lines 4180 and 4190 do a similar job on PED as in PEDESTAL. PEDESTAL is an easy word to mistype, and we can accept it on the basis of the first three letters since we know that there are no other words around with the same first three letters. It's nice to provide this little service to the player, whose sticky fingers may be burning such a hole in the computer as not always to get the right thing down, even though, God bless him, he was close. So we give him the benefit of the doubt, and good luck to him.

The other thing out of the ordinary is lines 4310–4370. This, as you can see, is the message from our friends the Krenn. Here we've combined some description from our puppet robot with the all capitalized Krenn transmission. Not bad, if I do say so myself. The GOSUB 63000 refers to the small pause you've already set in CONWCOM.

Finally, line 4380 cowcatches everything that fell through the cracks up to here, printing A$(9) and sending us back to 3000, where it all began.

THREE-LETTER WORDS

We're starting to run out of new tricks in our adventure programming bag, which means in a way that we are becoming old dogs. We will look at a few more little things that make the adventure design easier overall, but we've now covered all the major concepts for the design of this kind of game. Not that we've covered all there is to know about game programming—the poker program contains dozens of cans of all-new worms—and for that matter we're not even near the end of filling in ''Space Derelict!'' But we are over the hump as far as the technical concepts are concerned. Let's run the three-letter words up the flagpole and see who salutes.

''Space Derelict!''

```
4390   REM  THREE LETTER WORDS
4400   IF C$ = "RUN" THEN PRINT "I need a d
         irection."
     : GOTO 3000
4410   IF LEFT$ (C$,3) = "RUN" THEN C$ = "G
         O" + MID$ (C$,4)
     : GOTO 3000
4420   IF C$ = "CRY" THEN PRINT "Series R R
         obots are designed not to cry as a
          preventative against rust."
     : GOTO 3000
4430   IF LEFT$ (C$,3) = "GET" THEN C$ = "T
         AKE" + MID$ (C$,4)
     : GOTO 3110
4440   IF LEFT$ (C$,3) = "TIE" AND CO(12)
         = 1 THEN PRINT "I don't see much p
         oint to that."
     : GOTO 3000
```

```
4450   IF LEFT$ (C$,3) = "TIE" THEN PRINT A
         $(8)
     : GOTO 3000
4460   IF C$ = "SAY" THEN PRINT A$(1)
     : GOTO 3000
4470   IF C$ < > "SAY OPEN" THEN 4720
4480   PRINT "Okay."
     : PRINT "OPEN"
4490   IF CO(3) < > 1 THEN PRINT "Nothing h
         appens."
     : GOTO 3000
4500   IF R = 6 AND D(1) = 0 THEN D(1) = 1
     : PRINT "The northern door has opened.
         "
     : GOTO 3000
4510   IF R = 7 AND D(2) = 0 THEN D(2) = 1
     : PRINT "The north door has opened."
     : GOTO 3000
4520   IF R = 11 THEN D(11) = 1
     : PRINT "I'm willing to go in if you t
         hink it's advisable."
     : GOTO 3000
4530   IF R = 10 AND D(3) = 1 AND D(4) = 1
         THEN PRINT "Nothing happens."
     : GOTO 3000
4540   IF R = 10 AND D(4) = 0 THEN D(4) = 1

     : PRINT "The north door has opened."
4550   IF R = 10 AND D(3) = 0 THEN D(3) = 1

     : PRINT "The south door has opened."
4560   IF R = 10 THEN 3000
4570   IF R = 22 AND D(4) = 0 THEN D(4) = 1

     : PRINT "The south door has opened."
     : GOTO 3000
4580   IF R = 16 AND D(16) = 0 THEN D(16)
         = 1
     : PRINT "The east door has opened."
     : GOTO 3000
4590   IF R = 17 THEN PRINT "All the doors
         are open."
     : FOR X = 14 TO 17
     : D(X) = 1
     : NEXT X
     : GOTO 3000
4600   IF R = 13 AND D(15) = 0 THEN D(15)
         = 1
     : PRINT "The door is now open."
     : GOTO 3000
4610   IF R = 14 AND D(14) = 0 THEN D(14)
         = 1
     : PRINT "The door is now open."
     : GOTO 3000
4620   IF R < > 15 THEN 4680
4630   IF D(16) = 0 THEN D(16) = 1
     : PRINT "The northwest door has opened
         ."
```

```
4640  IF D(15) = 0 THEN D(15) = 1
     : PRINT "The southwest door has opened
        ."
4650  IF D(17) = 0 THEN D(17) = 1
     : PRINT "The northeast door has opened
        ."
4660  IF D(14) = 0 THEN D(14) = 1
     : PRINT "The southeast door has opened
        ."
4670  PRINT "Okay."
     : GOTO 3000
4680  IF R = 17 AND D(17) = 0 THEN D(17)
        = 1
     : PRINT "The west door is now open."
     : GOTO 3000
4690  IF R = 18 AND D(6) = 0 THEN D(6) = 1

     : PRINT "The south door is now open."
     : GOTO 3000
4700  IF R = 19 AND D(6) = 0 THEN D(6) = 1

     : PRINT "The door is now open."
     : GOTO 3000
4710  PRINT "Nothing happens"
     : GOTO 3000
4720  IF LEFT$ (C$,3) = "SAY" THEN PRINT "
        Okay."
     : PRINT MID$ (C$,4)
     : PRINT "Nothing happens."
     : GOTO 3000
4730  IF R = 22 AND CO(12) = 0 AND CO(9)
        = 1 AND C$ = "CUT ROPE" THEN
        PRINT "Okay."
     : CO(12) = 122
     : GOSUB 13000
     : GOTO 3000
4740  IF C$ = "CUT ROPE" AND CO(9) < > 1
        THEN PRINT A$(6)
     : GOTO 3000
4750  IF C$ = "CUT ROPE" AND (CO(12) < > 1
        00 + R OR (R < > 22 AND CO(12) = 0
        ) OR CO(12) < > 1) THEN PRINT A$(1
        )
     : GOTO 3000
4760  IF C$ = "CUT" THEN PRINT A$(1)
     : GOTO 3000
4770  IF LEFT$ (C$,3) = "CUT" THEN PRINT "
        I cannot cut that."
     : GOTO 3000
4780  PRINT A$(2) + LEFT$ (C$,3)
     : GOTO 3000
```

This time we'll concentrate just on special cases.
In ''Space Derelict!'' there is no advantage to the player in going some-

where quickly, so I've relegated the word RUN to synonym stature in line 4410. Since we've already excluded RUN as a complete command in 4400, we know that if RUN is the first word, then the length of C$ is greater than 3. So line 4410 transmutes the RUN command into a GO command by redefining C$ as GO plus all the rest of the original C$ starting with the space. Then the program branches to 3840, which is the beginning of the two-letter first-word commands, since that exactly defines our re-created C$. The more synonyms you have, the more user-friendly your game, and as you can see there's not much work in it for you as the designer.

Line 4420 is another joke. I point this out because it might look like it, but a joke it is. If your jokes are any better, let me know.

Line 4430 is another synonym, and I only mention it because I have played games where TAKE was not an acceptable command, which amazes me because as a player it took me a long time to come up with GET as a substitute. But apparently it is not unusual for some adventurers to operate the other way around entirely, and it never occurs to them to TAKE anything. In any case, both commands are essential in your program.

Line 4440 introduces the "intelligent" element of our puppet robot once again. Since in our game concept the robot possesses some sort of native intelligence, it makes sense that he perform some actions on his own (as with turning on the flashlight in room 23) and also that he occasionally has a chance to get his two cents in. This concept is fairly nascent in "Space Derelict!" and is one you can develop quite a bit further in an original game to make quite an interesting variation on your basic adventure. Line 4450, of course, works from the logical exclusions of 4440.

The use of the command SAY in your adventure is entirely optional, but magic words and the like come up often enough to have made it worth including an arrangement of this sort in "Space Derelict!" Since saying the word OPEN is the clincher in this game, you can see how this command has its effect in lines 4480–4700. Line 4710 acts as a cowcatcher from the SAY OPEN routine and 4720 acts a catchall for any game where SAYing something is going to have no effect. Line 4720 comes back with an OKAY and then prints C$ starting with the space so that the player sees repeated on the screen what he wanted to say. So if the player commands:

```
SAY ABRACADAVER
```

the computer replies with

```
OKAY
ABRACADAVER
NOTHING HAPPENS
```

and back we go to the beginning of 3000. When we say here that nothing happens, we really mean it, but it looks good, and it may fool the player into saying everything that comes into his head, which will be one more step each time on the road away from solving your puzzle, which is something we want to see as often as possible.

Within the SAY OPEN section there is an interesting little phenomenon in lines 4530–4560. If the player is in room 10 and says the magic word, there are two doors, D(4) and D(3), that may or may not respond to him. If D(4) is closed, the command will open that door, and the player will be so notified in line 4540. Then the processing continues to 4550. If D(3) is closed, then this door too will open and again the player will be so notified.

Line 4560 sends him back to 3000. As you can see, there are four possibilities here, depending on the values of D(4) and D(3). The player will see on the screen that one has opened, the other has opened, both have opened, or neither have opened, depending on the situation. This kind of programming structure allows for multiple possibilities where such are desirable, and the nice part of it is, they appear to occur virtually simultaneously on the video monitor.

Line 4750 contains a nasty bit of Boolean operation of parentheses within parentheses that really makes for an interesting situation. Broken down into its component parts, it looks like this:
IF AND (OR (AND) OR) THEN
Once again we're looking for truth. First the computer will determine the truth of the innermost AND. Those mean-looking ORs allow that if any one of the statements within these next parentheses is true, then the whole parenthetical statement becomes true. So if both of the halves of the innermost AND are true, or if either of the other OR possibilities within the parentheses is true, than the whole parenthetical statement gets ranked as truth. Now all that has to be determined is if both halves of the outermost AND contain truth. If so, then processing will proceed along this line. What we're trying to do in this line could have been accomplished in three lines, but the use of Boolean logic allows us to squeeze it all into one. I've been relatively spare with this sort of programming in "Space Derelict!" because it obviously can become very sticky very easily. It's not the easiest thing in the world to explain in the abstract, and more to the point, it makes it harder to follow what's going on in the program. But the more of this kind of programming you can do yourself, the more efficient your program will be, leaving more space for nastiness in your complications rather than complications in your programs.

A FEW CHOICE FOUR-LETTER WORDS

No-man's-land continues now, with line after line of programming drudgery. But each line serves a purpose, and once you have a list of all the words you want to use in your adventure, you'll be surprised how quickly this phase of the programming can actually go. With one exception, all of the four-letter words below are rated G.

"Space Derelict!"

```
4790  REM  4 LETTER WORDS
4800  S$ = LEFT$ (C$,4)
4810  IF LEN (C$) > 4 THEN S2$ = MID$ (C$,
        5)
4820  IF S$ = "READ" THEN PRINT "There is
        nothing readable here."
     : GOTO 3000
4830  IF S$ = "PUSH" THEN C$ = "PRESS" + S
        2$
     : GOTO 3120
```

```
4840  IF S$ < > "TAKE" THEN 5240
4850  IF C$ = "TAKE" THEN PRINT A$(9)
    : GOTO 3000
4860  IF LEFT$ (C$,8) = "TAKE INV" THEN 13
        100
4870  IF S$ = "TAKE" AND WT = 5 THEN
        PRINT "I can't do that. I am carry
        ing too much. Do you wish an inven
        tory?"
    : GOTO 3000
4880  GOTO 4900
4890  WT = WT + 1
    : PRINT "Okay."
    : GOTO 3000
4900  IF S2$ = " METAL" AND CO(1) = 100 +
        R THEN CO(1) = 1
    : GOTO 4890
4910  IF C$ = "TAKE METAL" THEN PRINT A$(7
        )
    : GOTO 3000
4920  IF S2$ = " PENTAGON" AND CO(2) = 100
        + R THEN CO(2) = 1
    : GOTO 4890
4930  IF C$ = "TAKE PENTAGON" THEN PRINT A
        $(7)
    : GOTO 3000
4940  IF S2$ = " CRYSTAL" AND CO(3) = 100
        + R THEN CO(3) = 1
    : GOTO 4890
4950  IF C$ = "TAKE CRYSTAL" THEN PRINT A$
        (7)
    : GOTO 3000
4960  IF S2$ = " WEAPON" AND CO(4) = 100
        + R THEN CO(4) = 1
    : GOTO 4890
4970  IF C$ = "TAKE WEAPON" THEN PRINT A$(
        7)
    : GOTO 3000
4980  IF S2$ = " BOX" AND CO(5) = 100 + R
        THEN CO(5) = 1
    : GOTO 4890
4990  IF C$ = "TAKE BOX" THEN PRINT A$(7)
    : GOTO 3000
5000  IF S2$ = " STAR" AND CO(6) = 115
        AND CO(1) = 1 AND R = 15 THEN CO(6
        ) = 1
    : PRINT "Okay."
    : GOTO 4890
5010  IF S2$ = " STAR" AND CO(6) = 100 + R
        AND R < > 15 THEN CO(6) = 1
    : PRINT "Okay."
    : GOTO 4890
5020  IF S2$ = " STAR" AND CO(6) = 100 + R
        THEN PRINT "It's attached somehow.
         I need something to loosen it."
    : GOTO 3000
```

```
5030  IF S2$ = " STAR" THEN PRINT A$(7)
    : GOTO 3000
5040  IF S2$ = " LASER" AND CO(7) = 100 +
        R THEN CO(7) = 1
    : GOTO 4890
5050  IF C$ = "TAKE LASER" THEN PRINT A$(7
        )
    : GOTO 3000
5060  IF LEFT$ (S2$,5) = " FLAS" AND CO(8)
        = 100 + R THEN CO(8) = 1
    : GOTO 4890
5070  IF LEFT$ (C$,11) = "TAKE FLASHL"
        THEN PRINT A$(7)
    : GOTO 3000
5080  IF S2$ = " KNIFE" AND CO(9) = 100 +
        R THEN CO(9) = 1
    : GOTO 4890
5090  IF C$ = "TAKE KNIFE" THEN PRINT A$(7
        )
    : GOTO 3000
5100  IF S2$ = " BUCKET" AND CO(10) = 100
        + R THEN CO(10) = 1
    : GOTO 4890
5110  IF S2$ = " BUCKET" AND CO(15) = 100
        + R THEN CO(15) = 1
    : GOTO 4890
5120  IF C$ = "TAKE BUCKET" THEN PRINT A$(
        7)
    : GOTO 3000
5130  IF S2$ = " ROBOT" AND CO(11) = 100
        + R THEN CO(11) = 1
    : GOTO 4890
5140  IF S2$ = " ROBOT" AND CO(13) = 100
        + R THEN PRINT A$(1)
    : GOTO 3000
5150  IF S2$ = " ROBOT" AND CO(16) = 100
        + R THEN CO(16) = 1
    : GOTO 4890
5160  IF C$ = "TAKE ROBOT" THEN PRINT A$(1
        )
    : GOTO 3000
5170  IF S2$ = " ROPE" AND CO(12) = 100 +
        R THEN CO(12) = 1
    : GOTO 4890
5180  IF S2$ = " ROPE" AND CO(12) = 0 AND
        R = 22 THEN PRINT A$(8)
    : GOTO 3000
5190  IF C$ = "TAKE ROPE" THEN PRINT A$(7)
    : GOTO 3000
5200  IF C$ = "TAKE WATER" AND R = 22 AND
        CO(14) = 122 AND CO(10) = 1 THEN C
        O(10) = 0
    : CO(15) = 1
    : PRINT "Okay."
    : GOSUB 13000
    : GOTO 3000
```

```
5210   IF C$ = "TAKE WATER" AND R = 22 AND
          CO(14) = 122 THEN PRINT A$(1)
     : GOTO 3000
5220   IF C$ = "TAKE WATER" THEN PRINT A$(6
          )
     : GOTO 3000
5230   PRINT "I cannot " + C$
     : GOTO 3000
5240   IF C$ = "SAVE" OR C$ = "SAVE GAME"
          THEN 30000
5250   IF S$ < > "QUIT" THEN 5290
5260   INPUT "Do you wish to try again";C$
5270   IF C$ = "Y" OR C$ = "YES" THEN RUN "
          INTRO"
5280   IF C$ = "N" OR C$ = "NO" THEN END EL
          SE5550
5290   IF C$ = "FUCK YOU" THEN PRINT "Pleas
          e maintain a proper attitude. The
          fate of the earth rests in your ha
          nds."
     : GOTO 3000
5300   IF S$ = "KICK" THEN PRINT A$(1)
     : GOTO 3000
5310   I, S$ = "HELP" THEN PRINT "Perhaps w
          e are not manipulating the items a
          t hand correctly."
     : GOTO 3000
5320   IF C$ = "KILL ROBOT" AND R = 23
          THEN PRINT A$(8)
     : GOTO 3000
5330   IF S$ = "KILL" THEN PRINT A$(1)
     : GOTO 3000
5340   IF S$ < > "OPEN" THEN 5820
5350   IF C$ = "OPEN BOX" AND (CO(5) = 1
          OR CO(5) = 100 + R) AND CO(6) = 5
          THEN PRINT "The crystal star is in
           it."
     : GOTO 3000
5360   IF C$ = "OPEN BOX" AND (CO(5) = 1
          OR CO(5) = 100 + R) THEN PRINT "Ok
          ay."
     : GOTO 3000
5370   IF C$ = "OPEN BOX" THEN PRINT A$(7)
     : GOTO 3000
5380   IF C$ < > "OPEN DOOR" THEN PRINT A$(
          1)
     : GOTO 3000
5390   IF R = 1 OR R = 11 THEN PRINT "There
           is no door here."
     : GOTO 3000
5400   IF R < 4 THEN PRINT A$(8)
     : GOTO 3000
5410   IF R < 6 OR R = 8 OR R = 9 OR R = 20
          THEN PRINT "There is no door here
          to open."
     : GOTO 3000
```

```
5420  IF R = 6 AND D(1) = 1 AND D(2) = 1
        THEN PRINT "The doors here are ope
        n."
    : GOTO 3000
5430  IF R = 6 THEN PRINT A$(8)
    : GOTO 3000
5440  IF R = 7 AND D(2) = 0 THEN PRINT A$(
        8)
    : GOTO 3000
5450  IF R = 7 THEN PRINT "The door is alr
        eady open."
    : GOTO 3000
5460  IF R = 10 AND D(4) = 0 AND D(3) = 0
        AND D(13) = 0 THEN PRINT A$(8)
    : GOTO 3000
5470  IF R = 10 AND D(4) = 1 THEN PRINT "T
        he north door is open."
5480  IF R = 10 AND D(3) = 1 THEN PRINT "T
        he south door is open."
5490  IF R = 10 THEN 3000
5500  IF R = 12 AND D(3) = 0 THEN PRINT A$
        (8)
    : GOTO 3000
5510  IF R = 12 THEN PRINT "The door is op
        en."
    : GOTO 3000
5520  IF R = 13 AND D(15) = 0 THEN PRINT A
        $(8)
    : GOTO 3000
5530  IF R = 13 THEN PRINT "The door is op
        en."
    : GOTO 3000
5540  IF R = 14 AND D(14) = 0 THEN PRINT A
        $(8)
    : GOTO 3000
5550  IF R = 14 THEN PRINT "The door is op
        en."
    : GOTO 3000
5560  IF R < > 15 THEN 4610ELSE FOR X = 14
        TO 17
5570  IF D(X) = 0 THEN PRINT A$(8)
    : X = 17
    : FLAG = 1
5580  NEXT X
5590  IF FLAG = 1 THEN FLAG = 0
    : GOTO 3000
5600  PRINT "All the doors here are open."
    : GOTO 3000
5610  IF R = 16 AND D(16) = 0 THEN PRINT A
        $(8)
    : GOTO 3000
5620  IF R = 16 THEN PRINT "The door is op
        en."
    : GOTO 3000
5630  IF R = 17 AND D(17) = 0 THEN PRINT A
        $(8)
    : GOTO 3000
```

```
5640  IF R = 17 THEN PRINT "The door is op
        en."
      : GOTO 3000
5650  IF R = 18 AND D(5) = 0 AND D(6) = 0
        THEN PRINT A$(8)
      : GOTO 3000
5660  IF R = 18 AND D(5) = 1 AND D(6) = 1
        THEN PRINT "The doors here are bot
        h open."
      : GOTO 3000
5670  IF R = 18 AND D(5) = 1 AND D(6) = 0
        THEN PRINT "The south door is alre
        ady open. How should I open the no
        rth door?"
      : GOTO 3000
5680  IF R = 18 THEN PRINT "The north door
         is open. How should I open the so
        uth one?"
      : GOTO 3000
5690  IF R = 19 AND D(6) = 0 THEN PRINT A$
        (8)
      : GOTO 3000
5700  IF R = 19 THEN PRINT "The door is al
        ready open."
      : GOTO 3000
5710  IF R = 21 AND D(7) = 0 AND D(5) = 0
        THEN PRINT A$(8)
      : GOTO 3000
5720  IF R = 21 AND D(7) = 1 AND D(5) = 1
        THEN PRINT "The doors here are alr
        eady open."
      : GOTO 3000
5730  IF R = 21 AND D(7) = 1 THEN PRINT "T
        he north door is open. How do I op
        en the south one?"
      : GOTO 3000
5740  IF R = 21 THEN PRINT "The south door
         is open. How do I open the north
        one."
      : GOTO 3000
5750  IF R = 22 AND D(8) = 0 AND D(4) = 0
        THEN PRINT A$(8)
      : GOTO 3000
5760  IF R = 22 AND D(8) = 1 AND D(4) = 1
        THEN PRINT "The doors are already
        open."
      : GOTO 3000
5770  IF R = 22 AND D(8) = 1 THEN PRINT "T
        he north door is open. How do I op
        en the south door?"
      : GOTO 3000
5780  IF R = 22 THEN PRINT "The south door
         is open. How do I open the north
        door?"
      : GOTO 3000
```

```
5790  IF R = 23 AND D(8) + D(9) + D(7) = 3
        THEN PRINT "All the doors in this
        room are open."
     : GOTO 3000
5800  IF R = 23 THEN PRINT A$(8)
     : GOTO 3000
5810  IF R = 24 THEN PRINT "The door is al
        ready open."
     : GOTO 3000
5820  IF S$ < > "LOOK" THEN 6570
5830  IF C$ = "LOOK" THEN 50000
5840  IF S2$ = " AIRLOCK" AND (R = 1 OR R
        = 2) THEN PRINT A$(22)
     : GOTO 3000
5850  IF S2$ = " AIRLOCK" THEN PRINT A$(7)

     : GOTO 3000
5860  IF R = 1 AND S2$ = " ENTRANCE" THEN
        PRINT A$(22)
     : GOTO 3000
5870  IF S2$ = " ENTRANCE" THEN PRINT A$(7
        )
     : GOTO 3000
5880  IF S2$ = " SHIP" AND R = 1 THEN
        PRINT "It is enormous and five-sid
        ed."
     : GOTO 3000
5890  IF S2$ = " SHIP" THEN PRINT A$(7)
     : GOTO 3000
5900  IF S2$ = " PANEL" AND (R = 2 OR R =
        7 OR R = 15) THEN PRINT "It is six
         inches across. The buttons are no
        t labeled."
     : GOTO 3000
5910  IF S2$ = " PANEL" AND R = 24 THEN
        PRINT "I can add nothing to my ear
        lier description."
     : GOTO 3000
5920  IF S2$ = " PANEL" THEN PRINT A$(7)
     : GOTO 3000
5930  IF S2$ = " METAL" AND (CO(1) = 1 OR
        CO(1) = 100 + R) THEN PRINT A$(10)

     : GOTO 3000
5940  IF S2$ = " METAL" THEN PRINT A$(7)
     : GOTO 3000
5950  IF (S2$ = " ROBOT" OR S2$ = " OBJECT
        ") AND R = 11 THEN PRINT "I have n
        othing new to add."
     : GOTO 3000
5960  IF R = 24 AND S2$ = " SLOT" THEN
        PRINT "It is about 7 inches long."

     : GOTO 3000
5970  IF S2$ = " SLOT" THEN PRINT A$(7)
     : GOTO 3000
```

```
5980  IF S2$ = " OBJECT" AND R < > 11
        THEN PRINT A$(9)
     : GOTO 3000
5990  IF S2$ = " PENTAGON" AND (CO(2) = 1
        OR CO(2) = 100 + R) THEN PRINT A$(
        11)
     : GOTO 3000
6000  IF S2$ = " PENTAGON" THEN PRINT A$(7
        )
     : GOTO 3000
6010  IF S2$ = " CRYSTAL" AND (CO(3) = 1
        OR CO(3) = 100 + R) THEN PRINT A$(
        12)
     : GOTO 3000
6020  IF S2$ = " CRYSTAL" THEN PRINT A$(7)

     : GOTO 3000
6030  IF C$ = "LOOK AROUND" OR C$ = "LOOK
        ROOM" THEN 50000
6040  IF S2$ = " WEAPON" AND (CO(4) = 1
        OR CO(4) = 100 + R) THEN PRINT A$(
        13)
     : GOTO 3000
6050  IF S2$ = " WEAPON" THEN PRINT A$(7)
     : GOTO 3000
6060  IF S2$ = " HOLOGRAM" OR S2$ = " DEVI
        CE" THEN PRINT "I could tell you m
        ore if I were closer to it."
     : GOTO 3000
6070  IF S2$ = " PEDESTAL" AND R = 7 THEN
        PRINT "It is about four feet high.
         The crystal is seated on top of i
        t."
     : GOTO 3000
6080  IF S2$ = " PEDESTAL" AND R = 5 AND C
        O(5) = 105 THEN PRINT "It is about
         four feet high. That is where the
         box is."
     : GOTO 3000
6090  IF S2$ = " PEDESTAL" AND R = 5 THEN
        PRINT "It is about four feet high.
        "
     : GOTO 3000
6100  IF S2$ = " PEDESTAL" AND R = 15 AND
        CO(6) = 115 THEN PRINT "It is four
         feet high. That is where the star
         is."
     : GOTO 3000
6110  IF S2$ = " PEDESTAL" AND R = 15
        THEN PRINT "It is four feet high."

     : GOTO 3000
6120  IF S2$ = " PEDESTAL" THEN PRINT A$(7
        )
     : GOTO 3000
6130  IF S2$ = " WALL" THEN PRINT A$(22)
```

```
6140  IF S2$ = " BOX" AND (CO(5) = 1 OR CO
        (5) = 100 + R) THEN PRINT A$(14)
     : IF CO(6) = 5 THEN PRINT "The crystal
         star is in it."
6150  IF S2$ = " BOX" AND (CO(5) = 1 OR CO
        (5) = 100 + R) THEN 3000
6160  IF S2$ = " BOX" THEN PRINT A$(7)
     : GOTO 3000
6170  IF S2$ = " CEILING" AND R = 5 THEN
        PRINT "It's too high up for me to
        describe it in any greater detail.
        "
     : GOTO 3000
6180  IF S2$ = " CEILING" THEN PRINT A$(22
        )
     : GOTO 3000
6190  IF S2$ = " DEVICE" AND R = 5 THEN S2
        $ = " CEILING"
     : GOTO 6170
6200  IF S2$ = " PASSAGEWAY" THEN PRINT A$
        (22)
     : GOTO 3000
6210  IF S2$ = " FOOTPRINTS" AND (R = 8
        OR R = 9 OR R = 10) THEN PRINT A$(
        22)
     : GOTO 3000
6220  IF S2$ = " FOOTPRINTS" THEN PRINT A$
        (1)
     : GOTO 3000
6230  IF S2$ = " DUST" AND R = 8 THEN
        PRINT A$(22)
     : GOTO 3000
6240  IF S2$ = " DUST" THEN PRINT A$(1)
     : GOTO 3000
6250  IF LEFT$ (S2$,7) = " CROSSR" AND R
        = 9 THEN PRINT A$(22)
     : GOTO 3000
6260  IF LEFT$ (S2$,7) = " CROSSR" THEN
        PRINT A$(1)
     : GOTO 3000
6270  IF S2$ = " STAR" AND (CO(6) = 1 OR C
        O(6) = 100 + R) THEN PRINT A$(15)
     : GOTO 3000
6280  IF S2$ = " STAR" THEN PRINT A$(7)
     : GOTO 3000
6290  IF S2$ = " DOOR" THEN PRINT A$(22)
     : GOTO 3000
6300  IF S2$ = " BED" AND (R = 12 OR R = 1
        9) THEN PRINT "I'm not sure if it
        really is a bed, but that's my bes
        t guess."
     : GOTO 3000
6310  IF S2$ = " BED" THEN PRINT A$(7)
     : GOTO 3000
6320  IF LEFT$ (S2$,6) = " CHAIR" AND R =
        15 THEN PRINT A$(22)
     : GOTO 3000
```

```
6330  IF LEFT$ (S2$,6) = " CHAIR" THEN
        PRINT A$(7)
     : GOTO 3000
6340  IF ( LEFT$ (S2$,7) = " COMPUT" OR
        LEFT$ (S2$,7) = " CONSOL") AND R
        = 15 THEN PRINT A$(22)
     : GOTO 3000
6350  IF LEFT$ (S2$,7) = " COMPUT" OR
        LEFT$ (S2$,7) = " CONSOL" THEN
        PRINT A$(7)
     : GOTO 3000
6360  IF S2$ = " BUTTONS" AND (R = 15 OR R
        = 2 OR R = 7) THEN PRINT A$(22)
     : GOTO 3000
6370  IF S2$ = " BUTTONS" THEN PRINT A$(7)

     : GOTO 3000
6380  IF S2$ = " LASER" AND (CO(7) = 1 OR
        CO(7) = 100 + R) THEN PRINT A$(23)

     : GOTO 3000
6390  IF S2$ = " LASER" THEN PRINT A$(7)
     : GOTO 3000
6400  IF S2$ = " FLASHLIGHT" AND (CO(8) =
        1 OR CO(8) = 100 + R) THEN PRINT A
        $(16)
     : GOTO 3000
6410  IF S2$ = " FLASHLIGHT" THEN PRINT A$
        (7)
     : GOTO 3000
6420  IF S2$ = " KNIFE" AND (CO(9) = 100
        + R OR CO(9) = 1) THEN PRINT A$(17
        )
     : GOTO 3000
6430  IF S2$ = " KNIFE" THEN PRINT A$(7)
     : GOTO 3000
6440  IF R = 22 AND (S2$ = " COURT" OR
        LEFT$ (S2$,5) = " ROPE") THEN
        PRINT A$(22)
     : GOTO 3000
6450  IF S2$ = " COURT" OR LEFT$ (S2$,5)
        = " ROPE" THEN PRINT A$(7)
     : GOTO 3000
6460  IF S2$ = " POOL" AND R = 22 THEN
        PRINT "It is very cold on this shi
        p. Apparently the water in it has
        completely frozen. It is filled wi
        th ice."
     : GOTO 3000
6470  IF S2$ = " POOL" THEN PRINT A$(7)
     : GOTO 3000
6480  IF S2$ = " BUCKET" AND (CO(10) = 1
        OR CO(10) = 100 + R) THEN PRINT A$
        (18)
     : GOTO 3000
```

```
6490  IF S2$ = " BUCKET" AND (CO(15) = 1
        OR CO(15) = 100 + R) THEN PRINT A$
        (18)
     : GOTO 3000
6500  IF S2$ = " BUCKET" THEN PRINT A$(7)
     : GOTO 3000
6510  IF S2$ = " ROBOT" AND (CO(11) = 1
        OR CO(11) = 100 + R) THEN PRINT A$
        (19)
     : GOTO 3000
6520  IF S2$ = " ROBOT" AND (CO(13) = 1
        OR CO(13) = 100 + R) THEN PRINT A$
        (20)
     : GOTO 3000
6530  IF S2$ = " ROBOT" AND (CO(16) = 1
        OR CO(16) = 100 + R) THEN PRINT "I
        t is definitely out of commission.
         The water was effective."
     : GOTO 3000
6540  IF S2$ = " ROBOT" THEN PRINT A$(7)
     : GOTO 3000
6550  IF S2$ = " ROPE" OR S2$ = " WATER"
        THEN PRINT S2$ + " IS" + S2$
     : GOTO 3000
6560  PRINT "I am not certain what you mea
        n by the term" + S2$
     : GOTO 3000
6570  IF S$ < > "DROP" THEN 5760ELSE6950
6580  WT = WT - 1
     : GOSUB 13000
     : GOTO 3000
6590  IF S2$ = " METAL" AND CO(1) = 1
        THEN CO(1) = 100 + R
     : GOTO 6580
6600  IF S2$ = " PENTAGON" AND CO(2) = 1
        THEN CO(2) = 100 + R
     : GOTO 6580
6610  IF S2$ = " CRYSTAL" AND CO(3) = 1
        THEN CO(3) = 100 + R
     : GOTO 6580
6620  IF S2$ = " WEAPON" AND CO(4) = 1
        THEN CO(4) = 100 + R
     : GOTO 6580
6630  IF S2$ = " BOX" AND CO(5) = 1 THEN C
        O(5) = 100 + R
     : GOTO 6580
6640  IF S2$ = " STAR" AND CO(6) = 1 THEN
        CO(6) = 100 + R
     : GOTO 6580
6650  IF S2$ = " LASER" AND CO(7) = 1
        THEN CO(7) = 100 + R
     : GOTO 6580
6660  IF S2$ = " FLASHLIGHT" AND CO(8) = 1
        THEN CO(8) = 100 + R
     : GOTO 6580
```

```
6670   IF S2$ = " KNIFE" AND CO(9) = 1
         THEN CO(9) = 100 + R
     : GOTO 6580
6680   IF S2$ = " BUCKET" AND CO(10) = 1
         THEN CO(10) = 100 + R
     : GOTO 6580
6690   IF S2$ = " BUCKET" AND CO(15) = 1
         THEN CO(15) = 100 + R
     : GOTO 6580
6700   IF S2$ = " ROBOT" AND CO(11) = 1
         THEN CO(11) = 100 + R
     : GOTO 6580
6710   IF S2$ = " ROPE" AND CO(12) = 1
         THEN CO(12) = 100 + R
     : GOTO 6580
6720   IF S2$ = " ROBOT" AND CO(13) = 1
         THEN CO(13) = 100 + R
     : GOTO 6580
6730   IF S2$ = " WATER" AND CO(15) = 1
         THEN CO(10) = 1
     : PRINT "It freezes immediately."
     : CO(15) = 0
     : GOSUB 13000
     : GOTO 3000
6740   IF S2$ = " ROBOT" AND CO(16) = 1
         THEN CO(16) = 100 + R
     : GOTO 6580
6750   PRINT A$(1)
     : GOTO 3000
6760   IF S$ = "FIND" THEN PRINT A$(8)
     : GOTO 3000
6770   IF S$ < > "MELT" THEN 6820
6780   IF C$ = "MELT ICE" AND CO(7) = 1
         AND R = 22 THEN CO(14) = 100 + R
     : GOSUB 13000
     : GOTO 3000
6790   IF S2$ < > " ICE" THEN PRINT "I can'
         t melt " + S2$
     : GOTO 3000
6800   IF CO(7) < > 1 THEN PRINT A$(6)
     : GOTO 3000
6810   IF R < > 22 THEN PRINT "I don't see
         how I can do that successfully at
         this time."
     : GOTO 3000
6820   IF C$ = "LOCK DOOR" THEN PRINT "Ther
         e doesn't seem to be any way to do
          that. Apparently once they're ope
         n they stay open."
     : GOTO 3000
6830   IF S$ = "LOCK" THEN PRINT A$(1)
     : GOTO 3000
6840   PRINT A$(2) + S$
     : GOTO 3000
```

Right off the top we're going to add a new wrinkle that comes in handy when we have a lot of words in a particular length grouping, as we do with the four-letter ones. Line 4800 now gives us a variable we can use that will be the four-letter first word, just as S2$ (which is stylistically upgraded in 4810) gives us everything after the first word. If you're a curious sort and had been wondering where the 2 had come from in S2$, now you know.

For most of this section once again I won't go into every nook and cranny of every line, but I will point out the highlights. Line 4820, for instance, covers all C$ lengths; it doesn't matter what S2$ our player wants to read, because there is nothing readable on this spaceship, and that is that. The fact that the hologram is unreadable is readily available to the player, and that's the only item even remotely decipherable.

For the record, line 4860 is the branch to the TAKE INVENTORY routine at 13100. To cover typing errors, INVENTORY has been conveniently shortened to INV. This means, of course, that if the player enters TAKE INVITATION he will find himself with an inventory in his lap instead, but there are no similar words to INVENTORY in the game vocabulary, so we needn't worry about this. If our player is that far out, he's never going to get very far as an adventurer anyway.

Starting with 4870 we see how the weight problem is handled. You'll remember that our variable for weight is WT and that we decided to limit our puppet to carrying five items at any one time. So if WT=5, whenever our player enters TAKE anything, he's going to get a report that he is carrying too much. Even if he says TAKE REST or TAKE ASPIRIN, he's going to see that he's carrying too much. The program will analyze taking ability only when his WT has a value of less than 5.

In 4880 we skip to 4900, saving 4890 as the branch to go to whenever the player successfully takes a movable object. As you can see, this line increments WT by 1, and it shuttles back to 3000 for more conversation. Line 4900 demonstrates a successful TAKE, reassigning a value of 1 to CO(1), meaning, of course, that this item is now a part of inventory and branching to 4890 for weight incrementation. Line 4910 is the logical exclusion from 4900, pointing out that the given movable object is not around at this time and scooting back to 3000.

You'll remember that to remove the crystal star in room 15 the player needed to use the odd piece of metal originally planted in room 19. No special manipulation is required; simply having it on the puppet's person would do the job. Line 5000 shows how this is translated into the program, accounting for all the various items necessary and acting accordingly. Line 5010 is the default when the star is not in room 15. Once the star has been loosened there's no need of the tool anymore, and the star can simply be picked up if it has been left behind. Line 5020 handles the situation when the player is in room 15 with the star in place but without the odd piece of metal.

There is no reason for the variations in lines 5060 and 5070, except I typed it two ways to alleviate the boredom of the same old thing over and over again. Once more we give the player the benefit of the typing doubt.

The bunching up of robots in 5130 and the following lines is a result of our having two robots in the game (in addition to our puppet robot). There's the destroyed one in room 5 and the attacking one in room 23. These lines account for both varieties and differentiate between the active and deactivated attacking robot in room 23. Line 5200 refers to a situation similar to the star/odd piece of metal before: This time we need a bucket before we can take any water. Line 5230 is a special cowcatcher telling the player that the puppet cannot take whatever it was he wanted taken.

Line 5240 is the point from which we save a game. Line 5250 is how you quit, allowing you both to quit for good and simply to quit and start over again. If the latter is the case, all the variables are cleared when INTRO is run, but keep in mind that the first few lines of the program are structured so the player can play a saved game.

Line 5290 is wildly optional, but this is the sort of command most adventurers enter sooner or later.

Line 5310 introduces the subject of HELP. It is up to you as designer whether to provide player aids, and if so, how helpful those aids are going to be. As you can see, I have not included any in "Space Derelict!"; it's simply not that hard a game. Conversely, in a really hard game you might be defeating your purpose if you provide player help. But nonetheless if as a player you enjoy player aids, as a designer you might be inclined to put them in to suit your own tastes. The simplest way to handle a variety of HELPs—you'll probably want a number of them to cover a number of situations—is to organize them according to room number. So you might have a HELP routine something like this:

```
IF R = 1 THEN PRINT "Maybe a magic word might do the trick." :GOTO 3000
IF R = 5 THEN PRINT "Looked under any rocks lately?" :GOTO 3000
IF R = 825 THEN PRINT "Many a tear has to fall . . ." : GOTO 3000
PRINT "I'm afraid you're on your own here." :GOTO 3000
```

These lines would provide specific help for some of the rooms where you decide help might be necessary, and would simply flow through to the last line if the R value were not met, telling the player that help is not on its way, at least not in this instance.

Lines 5350–5360 are a variation on the star/pedestal business. This time we open a box and our puppet automatically reports an item in it if that item is there, and reports nothing if the box happens to be empty.

With a little bit of preplanning the section beginning at 5380 could have been reduced somewhat, but the way it's done shows the lazy man's approach, simply going room by room and answering the command OPEN DOOR.

Lines 5560–5600 bring up something we've avoided mentioning so far about FOR . . . NEXT loops: getting out of them. FOR . . . NEXT loops, as you know, require both halves to operate correctly; when we're doing a sort, after we've found what we're looking for we can't simply break out of the loop. We must take it to its final value. Whenever the computer encounters a NEXT without a FOR or vice versa the operation of the program is interrupted with an error message. What we're looking for in 5570 is a closed door from D(14) to D(17). If we get one we want to print A$(8) and break out of the loop. But a GOTO 3000 in line 5570 would give us a FOR WITHOUT NEXT error. So what we do to terminate the loop is set X at its maximum value, in this case 17, and then set a FLAG. This way we'll go to 5580 without having to go back through the loop again, and the programming in 5590 handles our FLAG situation. Actually, the final value of X after line 5580 would be 18. NEXT statements increment your variable by the STEP value, measuring against the parameters you've established until the top number of the FOR . . . NEXT loop is reached. The loop here would keep incrementing X by 1 (the default STEP value). So if X = 14, the first value when you hit the NEXT the program increments X by the STEP value, then compares X (now 15) *minus* the STEP value to the top number set at the beginning of the loop. At the end X will equal 17 and once more the NEXT is encountered. Once again X will be incremented by the STEP and compared to the original parameter. In this case X will equal 18. Since 18 − 1 = 17, the FOR . . . NEXT loop will be terminated and processing will continue on the next line—and X will still be equal to 18. Forgetting how the STEP value works can lead to trouble, as can branching out of loops in programs, so keep these in mind when you're rolling your own.

Beginning at 5820 is the LOOK section. As far as I can tell I have provided a description of every object I have mentioned in every single room, plus one or two implied objects. I mentioned this before but I'll repeat it now: It is of the utmost importance that your program respond to mentions of everything that exists in your adventure. If you describe a room as having venetian blinds, even if they are purely decorative and unmovable, you must be able to respond to player input referring to those blinds. If a player is in a room with those blinds clearly described on the screen, and GO BLINDS or TAKE BLINDS gets a blank response, then you've got a hole in your program. Successfully accounting for everything is sometimes an impossible feat—a stray item will occasionally get by the most diligent programmer, and even professional programs occasionally miss one or two pieces of the obvious furniture—but it is one you should try to accomplish at any cost.

As we said earlier, a plain "LOOK" command will branch to 50000 for a new room display. Again, this way the player can call back the descriptive data that may have scrolled away. 5830 and 6030 both perform this function. Zip now to line 6560, a variation on the theme of cowcatching, where within a given S$ section (in this case LOOK) we respond to unidentifiable input by printing the S2$ rather than the whole C$. This clues the player into the fact that his first word was correct but that he might as well retire the second word as we just don't understand it.

Line 6520 and following cover DROP, the opposite of TAKE. It does everything the other way around, subtracting 1 from WT rather than adding to it. And this time, all successful DROPs will branch off to 13000 before going back to 3000 for a new movable object display, with the recently dropped movable object now added to the collection in the room.

And that's about it on the four-letter words. Hang in there, pilgrim, we're almost through it.

THE REST OF THE CHITCHAT

This is the last batch of the conversation segment.

"Space Derelict!"

```
6850   REM  START 5 LETTER WORDS
6860   S$ = LEFT$ (C$,5)
     : IF LEN (C$) > 5 THEN S2$ = MID$ (C$,
        6)
6870   IF S$ < > "PRESS" THEN 7050
6880   IF R < > 2 THEN 6960
6890   IF S2$ = " BUTTON" THEN PRINT A$(9)
     : GOTO 3000
6900   IF S2$ = " BLACK" AND AL = 0 THEN AL
        = 1
     : PRINT "The airlock has closed behind
         me."
     : GOTO 3000
6910   IF S2$ = " BLACK" AND AL = 1 THEN
        PRINT A$(21)
     : GOTO 3000
6920   IF S2$ = " BLUE" THEN 16000
6930   IF S2$ = " GREEN" AND D(10) = 0
        THEN D(10) = 1
     : PRINT "The door into the ship has op
        ened."
     : GOTO 3000
6940   IF S2$ = " GREEN" THEN PRINT A$(21)
     : GOTO 3000
6950   PRINT A$(1)
     : GOTO 3000
6960   IF R < > 7 THEN 7010
6970   IF S2$ = " BLUE" AND D(2) = 1 THEN D
        (2) = 0
     : PRINT "The door has closed."
     : GOTO 3000
6980   IF S2$ = " BLACK" OR (S2$ = " BLUE"
        AND D(2) = 0) THEN PRINT A$(21)
     : GOTO 3000
6990   IF S2$ = " GREEN" THEN 16000
7000   PRINT A$(1)
     : GOTO 3000
7010   IF R = 24 THEN PRINT A$(21)
     : GOTO 3000
7020   IF R < > 15 THEN PRINT A$(7)
     : GOTO 3000
7030   IF LEFT$ (S2$,5) = " BUTT" OR S2$ =
        " PANEL" THEN PRINT A$(21)
     : GOTO 3000
7040   PRINT A$(1)
     : GOTO 3000
7050   IF LEFT$ (C$,10) = "CLIMB WALL"
        THEN PRINT "Is that a metaphorical
         statement? I cannot climb these w
        alls."
     : GOTO 3000
```

```
7060  IF S$ = "TOUCH" THEN S$ = "PRESS"
    : GOTO 6870
7070  IF S$ = "CLIMB" AND R = 22 AND CO(12
        ) = 0 THEN PRINT "Okay."
    : FOR PAUSE = 1 TO 1000
    : NEXT
    : GOTO 16000
7080  IF S$ = "CLIMB" THEN PRINT A$(1)
    : GOTO 3000
7090  IF (C$ = "ENTER" OR C$ = "ENTER SHIP
        ") AND R = 2 AND D(10) = 1 THEN R
        = 3
    : GOTO 50000
7100  IF (C$ = "ENTER" OR C$ = "ENTER SHIP
        ") AND R = 2 THEN PRINT A$(5)
    : GOTO 3000
7110  IF (C$ = "ENTER" OR C$ = "ENTER SHIP
        ") AND R = 1 THEN R = 2
    : GOTO 50000
7120  IF C$ = "ENTER AIRLOCK" AND R = 1
        THEN R = 2
    : GOTO 50000
7130  IF C$ = "ENTER" OR C$ = "ENTER SHIP"
        OR C$ = "ENTER AIRLOCK" THEN
        PRINT A$(1)
    : GOTO 3000
7140  IF S$ < > "LIGHT" THEN 7180
7150  IF LEFT$ (C$,9) < > "LIGHT FLA"
        THEN PRINT A$(1)
    : GOTO 3000
7160  IF CO(8) < > 1 THEN PRINT A$(7)
    : GOTO 3000
7170  L = 1
    : PRINT "Okay."
    : GOTO 50000
7180  IF C$ = "LEAVE" THEN PRINT "There's
        no turning back now."
    : GOTO 3000
7190  IF S$ = "LEAVE" THEN C$ = "DROP" +
        MID$ (C$,6)
    : GOTO 6590
7200  IF C$ = "THROW" THEN PRINT A$(9)
    : GOTO 3000
7210  IF C$ = "THROW WATER" AND CO(15) = 1
        AND R = 23 AND CO(13) = 123 THEN
        PRINT "It stopped him dead in his
        tracks."
    : CO(13) = 0
    : CO(16) = 123
    : GOSUB 13000
    : GOTO 3000
7220  IF C$ = "THROW WATER" AND CO(15) = 1
        THEN S$ = "LEAVE"
    : GOTO 7190
7230  IF S$ = "THROW" THEN PRINT A$(1)
    : GOTO 3000
```

```
7240  IF C$ = "SCORE" THEN PRINT "The busi
        ness at hand is the saving of the
        solar system. I will tell you when
         I  have successfully completed th
        at task."
    : GOTO 3000
7250  IF S$ = "LASSO" AND CO(12) = 1 THEN
        PRINT "Series R Robots are not qua
        lified to perform cowboy chores."
    : GOTO 3000
7260  IF S$ = "LASSO" THEN PRINT A$(8)
    : GOTO 3000
7270  IF S$ = "BLAST" THEN S$ = "SHOOT"
    : GOTO 7310
7280  IF C$ = "THINK" THEN PRINT "Think wh
        at?"
    : GOTO 3000
7290  IF S$ = "THINK" THEN C$ = "SAY" +
        MID$ (C$,6)
    : GOTO 4470
7300  IF S$ < > "SHOOT" THEN 7380
7310  IF C$ = "SHOOT" THEN PRINT A$(9)
    : GOTO 3000
7320  IF CO(7) < > 1 THEN PRINT "I am not
        carrying my laser pistol."
    : GOTO 3000
7330  IF S2$ = " ROBOT" AND CO(13) = 100
        + R THEN PRINT "It doesn't have an
        y effect! It doesn't even slow him
         down!"
    : GOTO 3000
7340  IF S2$ = " ICE" AND R = 22 THEN
        PRINT "Okay."
    : CO(14) = 100 + R
    : GOTO 50000
7350  IF S2$ = " DOOR" AND R = 23 AND D(9)
        = 0 AND CO(16) = 123 THEN PRINT "I
        t works. I can get through now."
    : D(9) = 1
    : GOTO 3000
7360  IF S2$ = " DOOR" AND R = 23 AND CO(1
        6) = 0 THEN 16000
7370  PRINT "Series R Robots are designed
        with maximum safety in mind. Becau
        se of this I      cannot " + C$
    : GOTO 3000
7380  IF C$ = "WHERE" THEN PRINT A$(1)
    : GOTO 3000
7390  IF S$ = "WHERE" THEN PRINT A$(7)
    : GOTO 3000
7400  IF (C$ = "CLICK" OR C$ = "CLICK PENT
        AGON") AND CO(2) < > 1 THEN
        PRINT "I'll have to have it in my
        hand first."
    : GOTO 3000
7410  IF C$ < > "CLICK" AND C$ < > "CLICK
        PENTAGON" THEN 7560
```

```
7420  IF R = 3 AND D(12) = 0 THEN D(12) =
        1
    : PRINT "The west door has opened."
    : GOTO 3000
7430  IF R = 6 AND D(2) = 0 THEN D(2) = 1
    : PRINT "The south door has opened."
    : GOTO 3000
7440  IF R = 18 AND D(5) = 0 THEN D(5) = 1

    : PRINT "The north door has opened."
    : GOTO 3000
7450  IF R = 21 AND D(7) = 1 AND D(5) = 1
        THEN PRINT A$(21)
7460  IF R = 21 AND D(5) = 0 THEN D(5) = 1

    : PRINT "The south door has opened."
7470  IF R = 21 AND D(7) = 0 THEN D(7) = 1

    : PRINT "The north door has opened."
7480  IF R = 21 THEN 3000
7490  IF R = 22 AND D(8) = 0 THEN D(8) = 1

    : PRINT " The north door has opened."
    : GOTO 3000
7500  IF R = 23 AND D(8) = 1 AND D(7) = 1
        THEN PRINT A$(21)
7510  IF R = 23 AND D(8) = 0 THEN D(8) = 1

    : PRINT "The southwestern door has ope
        ned."
7520  IF R = 23 AND D(7) = 0 THEN D(7) = 1

    : PRINT "The southeastern door has ope
        ned."
7530  IF R = 23 THEN 3000
7540  IF R = 10 THEN 16000
7550  PRINT A$(21)
    : GOTO 3000
7560  PRINT A$(2) + C$
    : GOTO 3000
7570  REM     START 6 LETTER WORDS HERE
7580  S$ = LEFT$ (C$,6)
    : IF LEN (C$) > 6 THEN S2$ = MID$ (C$,
        7)
7590  IF C$ = "INSERT" THEN PRINT A$(9)
    : GOTO 3000
7600  IF S$ < > "INSERT" THEN 7740
7610  IF R = 24 THEN 7670
7620  IF CO(5) < > 1 THEN PRINT "I have no
        thing to insert it in that I can s
        ee."
    : GOTO 3000
7630  IF S2$ = " METAL" OR S2$ = " PENTAGO
        N" OR S2$ = " WEAPON" THEN PRINT A
        $(1)
    : GOTO 3000
```

```
7640  IF S2$ = " CRYSTAL" AND CO(3) = 1
        THEN 16000
7650  IF S2$ = " STAR" AND CO(6) = 1 THEN
        CO(6) = 5
     : PRINT "It fits like a glove.I think
        that was a good idea, but I don't
        know why."
     : WT = WT - 1
     : GOTO 3000
7660  PRINT A$(1)
     : GOTO 3000
7670  IF S2$ = " BOX" THEN 7700
7680  IF S2$ = " STAR" AND CO(6) = 1 AND C
        O(5) = 1 THEN PRINT "I'll put it i
        nto the box... It fits like a glov
        e. I think that was a good idea."
     : WT = WT - 1
     : CO(6) = 5
     : GOSUB 13000
     : GOTO 3000
7690  PRINT A$(1)
     : GOTO 3000
7700  IF R = 24 AND CO(6) = 5 AND CO(5) =
        1 THEN 17000
7710  IF R = 24 AND CO(5) = 1 THEN 16000
7720  IF R = 24 THEN PRINT A$(7)
     : GOTO 3000
7730  PRINT A$(1)
     : GOTO 3000
7740  IF S$ = "UNLOCK" THEN PRINT A$(8)
     : GOTO 3000
7750  IF C$ = "REMOVE" THEN PRINT A$(9)
     : GOTO 3000
7760  IF C$ = "REMOVE STAR" AND CO(6) = 5
        THEN CO(6) = 100 + R
     : GOSUB 13000
     : GOTO 3000
7770  IF LEFT$ (C$,6) = "REMOVE" THEN
        PRINT A$(1)
     : GOTO 3000
7780  PRINT A$(2) + C$
     : GOTO 3000
7790  S$ = LEFT$ (C$,7)
7800  IF LEN (C$) > 7 THEN S2$ = MID$ (C$,
        8)
7810  IF S$ < > "UNLIGHT" THEN 7850
7820  IF (C$ = "UNLIGHT" OR LEFT$ (C$,12)
        = "UNLIGHT FLAS") AND L = 1 THEN
        PRINT "Okay"
     : L = 0
     : IF R = 23 THEN 50000ELSE3000
7830  IF (C$ = "UNLIGHT" OR LEFT$ (C$,12)
        = "UNLIGHT FLAS") THEN PRINT "The
        flashlight is not lit."
     : GOTO 3000
7840  PRINT A$(1)
     : GOTO 3000
```

```
7850   IF S$ = "EXAMINE" THEN C$ = "LOOK"
         + S2$
     : GOTO 5830
7860   PRINT A$(2) + C$
     : GOTO 3000
7870   IF LEFT$ (C$,8) = "DESCRIBE" THEN C$
         = "LOOK" + S2$
     : GOTO 5830
7880   PRINT A$(2) + C$
     : GOTO 3000
7890   IF LEFT$ (C$,5) = "INVEN" THEN 13100

7900   PRINT A$(2) + C$
     : GOTO 3000
```

We've covered almost every possible programming variation for this segment now, so most of this is just more of the same. But there are a couple of new twists. Line 6860, for example, finally puts all the S$/S2$ business on one line, making it as elegant and concise as possible. Now that you know what this is all about, this is the form you should use for every length variation.

The PRESS section covers all those buttons in rooms 2, 7, and 15 (with the synonyms TOUCH and PUSH also bringing us to this section). In 7040 you encounter ENTER, which is used as a substitute for GO at the beginning of the game where no EXIT directions are given. Covering such input as GO IN or ENTER in less demanding situations is entirely optional, but it is recommended in likely cases. Why shouldn't GO IN be a likely synonym for GO DOOR? The former is certainly more colloquial than the latter, at least in the real world, while the latter has its place really only in adventure lingo. While the designer must cover all the normal adventuring usages, there's nothing wrong with extending yourself as much as possible when there are perfectly good English phrases that can do what the perfectly good computer phrases also do. It's the old user-friendly concept; of course, user-friendly usually translates as "any idiot can understand it," but what's wrong with that? Accessibility is the name of the game, and you're not going to get very far as a game designer if you're the only one who can communicate with your programs.

Line 7240 brings up the business of providing a SCORE for the player as he progresses. In a situational adventure like "Space Derelict!" scoring is a non sequitur, since either you've defused the ship or you haven't. But in treasure-hunt adventures the score can keep the player apprised of how many treasures he's already collected vs. the total number of treasures he needs to win. There are two usual ways of handling this. The first is giving the player an artificial score from 0 to 100 based on the number of treasures he's already collected, simply by dividing the number of treasures he already has by the total number of treasures and multiplying the result by 100. That way, if there are 20 treasures, for instance, he would get five points for each one. A new variable such as TR would handle this for you:
IF C$ = "SCORE" THEN PRINT "OUT OF A POSSIBLE 100 YOU HAVE "; (TR/20) * 100

Of course, TR is the running count of the number of treasures collected, and in this hypothetical case 20 is the total number of treasures. The other way to handle it, again using TR as the variable count of collected treasures, is simply to say:
IF C$ = ``SCORE'' THEN PRINT ``OUT OF A POSSIBLE 20 TREASURES YOU HAVE '';TR
Whichever suits your fancy is perfectly appropriate. The value of TR would be established by totaling the number of treasures on hand in the treasure-storing room.

THINK in 7280 is the introduction of the telepathy factor, a very strange word indeed for an adventure. Line 7310 is a good example of a programming shortcut. As a rule, if you give your puppet a gun you have to accommodate his shooting everything in sight and would normally need a routine as long and complicated as the LOOKs were earlier. But by putting a little common sense into the robot's head we're able to bypass all the shoot-'em-up possibilities by allowing only for the shots fired that we wish to be fired and circumventing the rest by telling the player that the robot has more self-control than he does.

Lines 7380 and 7390 are not at all necessary for practiced adventurers, but I've never seen a newcomer who didn't try to get the computer to do the work of finding things for him by saying WHERE (WHATEVER) as if that would solve all his problems. So our user-friendly program includes it here. Line 7810 introduces the hideous word UNLIGHT, which is the only one I've heard of that will turn off the lantern or candle or whatnot in just one word. I'd really like to know if there are any better ones around. Line 7890 allows the bad typist to enter any variation he likes (provided the first five letters are correct) of the word INVENTORY to get same. And for those of you who never thought you'd see the end of this, line 7900 is the end of the conversation-with-the computer segment.

MISCELLANEOUS BUSINESS

We've now gone through 99.9 percent of all the programming necessary for a standard situational-type adventure, with a few side roads mentioned occasionally for the other kinds of adventures you might want to design. A few more tidbits like these will be discussed in Chapter 6, providing you with a basis for just about anything you might want to put into your games. The important point to keep in mind is one that was mentioned earlier and should become your designing motto: Almost everything can fit into the normal operation of the basic adventuring module, and one way or another you should *make* it fit. As you've seen in ``Space Derelict!'' we've handled everything from simple DROPs and TAKEs to INSERT, THROW, and BLAST, all in the conversation segment. But all these actions had one thing in common: They were all taken in response to input from the player. None of them was self-generating, which leaves us with the problem of finding a place to put those situations within the game where something will happen out of the player's control. In ``Space Derelict!'' this situation arises only once, when the player enters room 23, where the killer robot is waiting to pounce. Unless the player deactivates this robot or immediately leaves the room, the puppet robot will be destroyed after three turns. This is not as elementary as the trap in room 5, where all the player had to do was walk into the room to get zapped, and which was concisely handled in the GOs. This time the player has three moves before he enters Zap City, and we must accommodate him accordingly. Any miscellaneous business you come up with will be of a similar nature, be it roving monsters, a built-in clock limiting player moves, or whatever. You want to insinuate this material into the program so that it comes up only when you want it to, the way you want it to. The basic module gives you both a place to account for such activities and a place to program their details.

The place to account for such activities is at the beginning of the line 3000 conversation-with-the-computer segment in CONWCOM, *before* prompting the player input. Go back and look at line 3010: this is the one that handles the miscellaneous business in this game, and any number of other miscellaneous businesses also could have been inserted at this point, giving the necessary criteria and, if the variables fell within that criteria, branching out somewhere to handle the appropriate business. The reason this spot works so well for this is that all the subroutines do eventually branch back to 3000 and the player prompt, be they changes of room in 50000, or be they any actions covered in the conversation-with-the-computer section itself. All roads lead to 3000, so 3000 is the place for miscellany.

In "Space Derelict!" the criterion for getting involved in our miscellaneous business is simply our player being in room 23, where he would encounter the killer robot, so all line 3010 does is branch to 40000 on the condition that R=23. If R does not equal 23, then the program continues along completely unaffected. If R does equal 23, we hit the final programming lines of "Space Derelict!"

"Space Derelict!"

```
40000 REM  ROBOT
40010 IF CO(13)  = 0 THEN RETURN
40020 T = T + 1
40030 IF T = 1 THEN RETURN
40040 IF T = 2 THEN PRINT "The robot is at
        tacking me!"
    : RETURN
40050 IF T = 3 THEN 16000
```

This short subroutine handles the attack of the killer robot. You'll recall the unexplained T=0 statement in line 50110 in the room segment. This simply cleared the value of T for us so it wouldn't be hanging over our robot's head when he was out of room 23. T (for "time") is merely our counter for the number of moves to be taken in room 23. When the player enters room 23 and CO(13), the killer robot, is alive and well, the value of T is set at 1 by line 40020 (since T=0 to begin with, T+1=1). Then line 40030 returns us to 3010 and the rest of the conversation. The player knows the killer robot is in the room, of course, because it will be displayed along with any other movable objects in the room description.

Whatever move the player now makes—other than leaving room 23, which would get him out of the situation entirely—will bring him back to the beginning of 3000 and another run through the 40000 subroutine. Once again T will be incremented by 1, but this time T+1=2, so line 40040 comes into play, and the puppet reports that the killer robot is at his throat, and the play branches back to the conversation. Again, no matter what the player does, short of leaving room 23—or destroying the killer robot—he will branch back here again, and this time T will equal 3, meaning hello line 16000 and good-bye player. Once the killer robot is deactivated, however, all of this becomes moot per line 40010.

There is one hole in the way this is handled here, and it could have been sealed up, but if the player discovers it, it could make things more interesting for him. If you've figured out what that hole is, go to the head of the class. You see, if the player enters a LOOK command while he is under siege by the killer robot, the program will branch to 50000, resetting T at 0! So if the player wants to stand there in room 23 and keep LOOKing, he can do so until kingdom come and not get killed. But he won't progress any further in the game because the door to room 24 is sealed in more ways than one. That is, the player can't pull this trick to buy himself some extra time to blast that door, because there is a built-in fail-safe against such actions—go back and take a look at 7350–7360.

There is plenty of room between lines 40000 and 49900 to fit in any kind of miscellaneous business you want, but as you can see, there is one drawback to it. Every piece of miscellaneous business requires extra (and probably complicated) programming for successful execution. Everything else in the program works smooth as a goat, whereas treading into the waters of miscellany and 40000 can lead to entirely new problems. But not every complication is going to fit neatly into the prescribed pattern, so we have to have some place for the squeakers that just don't belong anywhere else, and at least now you know.

And believe it or not, we have now covered all of the "Space Derelict!" program, so if you have been entering it as it's been printed in the book, you now have the whole thing up and running—provided you've typed it all correctly. For the most part you now know everything you need to know to set out on your own adventure design. But writing your own adventures is not quite as easy as copying someone else's, so we have a little more ground to cover in the next couple of chapters.

5
BUGGING AND DEBUGGING

A RECAP OF THE ADVENTURE MODULE

Now we've gone bit by bit through an entire adventure, covering everything from the basic concept of what adventuring is all about, through the theories of programming techniques, right down to the nitty-gritty of the actual line programming. You should be ready to attack your own design at this point with a fair certainty of a reasonably running program when you're finished. But in this chapter I want to talk about something nobody ever seems to cover. When you read articles or books on computers, everybody tells you how to do everything right, showing you perfect programs and telling you how they run. But nobody ever tells you how that program got to be so perfect in the first place. How many drafts did that little ten-liner go through to get where it is today? By the time something gets to the publishing stage, not only is it debugged cleaner than a clerical whistle, but it's usually so polished that the *running* of the program becomes the end-all and be-all, not the understanding of *how* the program works. In many cases this makes for completely unintelligible—but nonetheless perfect—programs. No one will ever accuse us of that in "Space Derelict!" If you have a good eye for program stylization and streamlining (topics covered in Part Two of this book) you could probably shave 20 percent off the top of that one with no trouble at all. Rewrite large portions of it with all the action translated into variables and you could get out as much as 50 percent. But, of course, the program would lose much of its instructional benefit, which was the whole point of the exercise in the first place. Anyhow, what I want to discuss in this chapter is how "Space Derelict!" got to where it is vis-à-vis the actual entry process. There's more to writing a program than just writing a program, especially with adventures where we're dealing with four pieces of a basic module readjusted each time out to fit each new game. The module will remain the same, as will the programming and editing techniques, and these things are what we'll be talking about here.

First let's abstract the adventure module in a usable form so that you will have a basic outline for your own designs. This is the framework on which you and your adventures will hang (or be hanged, depending on the circumstances).

```
INTRO:

10 REM INTRO
20 SET DIMENSIONS
30 SET MORE DIMENSIONS
40 COMMON ALL YOUR PASSING VARIABLES
50 CLEAR SCREEN
60 GOSUB 15000: REM INTRODUCTION
70 CLEAR SCREEN
80 GOSUB 2000:REM INITIALIZATION
90–120 ALLOW FOR REPLAY OF SAVED GAME
130 CLEAR SCREEN
140–300 or so CONTINUE INTRODUCTION AS NECESSARY
310 CHAIN ''CONWCOM'', 49900, ALL
2000 INITIALIZATION/HOUSEKEEPING
2010 R = 1
2020–2250ish SET A$(X): REM ANSWERS FROM THE COMPUTER
2260–2410ish SET CO$(X): REM MOVABLE OBJECTS
2420–2450ish SET MISCELLANEOUS VARIABLES
2460 RETURN
15000–15200ish SET INTRODUCTION SUBROUTINE
31000 REPLAY SAVED GAME ROUTINE
31010 OPEN ''I'', 1, GAMEFILE$
31020–31120ish INPUT VARIABLES
31130 CLOSE 1
31140 PRINT ''Please stand by . . .''
31150 CHAIN ''CONWCOM'', 49900, ALL
```

```
LOSER:

16000 LOSING ROUTINE
16010–16100 TELL PLAYER IN NO UNCERTAIN TERMS THAT HE
   HAS LOST
16110 & FF ALLOW PLAYER TO START AGAIN ELSE END
63000 PAUSE
```

```
WINNER:

17000 WINNING ROUTINE
17010–17200 CONGRATULATE PLAYER FOR A JOB WELL DONE
17210 END
63000 PAUSE
```

```
CONWCOM:

3000 CONVERSATION WITH THE COMPUTER SEGMENT
3010–3050 SET MISCELLANEOUS BUSINESS PARAMETERS AND
   GOSUBS
3060 INPUT "PLAYER PROMPT";C$
3070–3200ish BREAK DOWN C$ BY SIZE AND GOTO APPROPRIATE
   GROUP
3210–wherever SET ACTUAL CONVERSATION
13000–13090 SET ROOM INVENTORY SUBROUTINE
13100–13210 SET PUPPET INVENTORY ROUTINE
16000 CHAIN "LOSER"
17000 CHAIN "WINNER"
30000 SAVE GAME
30010 OPEN "O", 1, "GAMEFILE:1"
30020–30110ish WRITE GAME VARIABLES INTO DISK FILE
30120 CLOSE 1
30130 PRINT "Game saved."
30140 GOTO 3000
40000–49000 MISCELLANEOUS BUSINESS ROUTINES AS NEEDED
49900 COMMON VARIABLES
50000 ROOM$ = "R" + STR$(R)
50010 IF MID$(ROOM$,2,1)=" " THEN ROOM$=LEFT$
   (ROOM$,1)+MID$(ROOM$,3)
50020 OPEN "I", 1, ROOM$
50030–50090 INPUT ROOM DESCRIPTION VARIABLES
50010 CLOSE 1
50020–50140ish PRINT R$(X)
50150 GOSUB 13000:REM MOVABLE OBJECT INVENTORY
50160–50200 PRINT N,S,E,W
50210 GOTO 3000
63000 PAUSE
```

And that's the whole ball game in its briefest form, with a lot of those numbers, of course, varying as the need arises.

GETTING IT INTO WORKABLE SHAPE

The nice thing about adventures is the comparative ease of programming. The not-nice thing about adventures is that each one is unique, with very few pieces that can be transported 100 percent intact from one adventure to the next. A few pieces can travel without variation, but for the most part everything depends on the demands of your game universe—and your game universe changes from adventure to adventure (or else you've taken up the wrong hobby). With strategy games this is not necessarily the case. Compare bridge, poker, and gin rummy, for example; they may be radically different in many ways, but at least they all revolve around cards, and programs for these games can contain identical shuffling routines, plus only slightly modified dealing routines and (if you use them) graphics. Another strike against adventures in pure programming terms is their lack of structure. For the most part all we're doing is going around and around the line 3000 conversation-with-the-computer segment, with occasional side trips to 50000 or 13000 or 13100. What we have here is more an extended loop than a program, and this does restrict us somewhat in that we can't take the program apart and analyze it bit by bit in our debugging stage; we're stuck with one enormous monster to be attacked *in toto*.

This means that you can't begin any serious debugging until you've finished the entire program. The point of this section is to provide you with some program-writing techniques that may save time when you reach the other end of the rainbow and have to start making your brainchild run. And the first rule of programming is one I'm sure you've heard over and over again: SAVE your work often and well. If you have to get up to go to the bathroom, SAVE what you've done so far. If there's a fire in the basement, SAVE your work before you grab the extinguisher. If the nuclear plant next store seems to be giving off a strange purple glow, SAVE your program before going to the window to get a better look. If you live alone in a garret this may not be much of a problem, but if any animal, human or otherwise, young or old, shares your living quarters with you and your computer, sooner or later he, she, or it is going to discover some unique and amazing way to delete seven hours of your hard work without even realizing it.

My favorite debugging command is RENUM, occasionally used in tandem with MERGE. RENUM allows us to adjust programs to let more air between the lines, and MERGE allows us to take a small piece of a big program, renumber it, then MERGE it back into the big program, which has its benefits. When I program and debug I find renumbering one of my most important tools, allowing me more room when I need it and closing up the gaps behind me when I don't. There's something about the continual neatness of 10-increment lines that just keeps me going when the going gets tough: Renumbering keeps me in 10-increment heaven.

I have mentioned before various bits and pieces of the entry process, so bear with me if I repeat myself here. The first thing to do—after you've designed your game and made your map and whatnot—is enter the two short routines at 13000 and 13100 in CONWCOM. At this stage you should know how many movable objects you have, and you can therefore enter the correct numbers into these routines where they are called for. If for any reason the number of movable objects changes along the way (and this isn't as unlikely as it sounds), don't forget to come back here and revise these numbers. You can take advantage of AUTO as you enter. AUTO is short for automatic numbering, done for you by the computer every time you Enter a new line, thus saving you the bother of having to type in line numbers. A plain AUTO starts you at line 10; AUTO 13000 would start you at line 13000. The normal increment between lines is 10. AUTO 13000, 20 would give you an increment of 20 starting at line 13000, while AUTO, 20 would give you a 20 increment starting at line 10. The comma tells the computer whether you mean starting line or increment.

Next, use RMAKER to create the rooms, followed by putting the 50000 room layout routine into CONWCOM. After entering all the descriptions, you're ready to print them. As I recommended earlier, you might find it best to work from a printout of the room-layout routine. Next, enter the pause in CONWCOM. I prefer the upper reaches because the whole point of a pause is to slow something down, and what could be slower than dredging through a whole program to find line 63000? Must take a whole microsecond, give or take a fraction or two.

My next step (by the way, although this sounds as though I'm sitting there zipping through all this, we're actually talking about quite a few hours of work here) is to program INTRO, the starting lines of the game. At this point you may be a bit fuzzy about how large an array to set for A$ and perhaps even the CO family, so do the best you can and keep in mind that these lines may need changing later. Some of this section will vary from game to game, while some is fixed, such as the branches to 15000 and 2000 and the replay-saved-game prompt. As you continue the introduction you'll be doing some actual creative composition, and you might want to work some of this out on paper first if you feel more relaxed writing that way. Whichever you prefer, writing at the keyboard or on good old-fashioned paper, when you've finished entering the entire introduction you are now going to call on Renumber. Let's say your introduction stretches from line 10 to line 250. Simply type in RENUM,, 20 and the command will automatically renumber the lines from 10 to 250 in increments of 20 rather than the increments of 10. RENUM works like this—RENUM X,Y,Z—with X, Y, and Z all being optional. X is the new starting number, Y is the number in the old program where renumbering should begin, and Z is the increment between lines. As with AUTO, the commas tell the whole story. You could, of course, number in increments of 20 in the first place, but you get so used to increments of 10 that it's hard to think in terms of anything else. The reason for allowing so much space between lines is simple: We're preparing now for the errors to come, allowing space for addenda that will no doubt be necessary as we start our debugging/revising frenzy. It might not be absolutely essential in some of the sections of the program, but when we get to the conversation-with-the-computer segment this extra space will be our guiding light, our prayer, and our salvation. And it doesn't hurt to leave space in the other sections as well where there's a possibility it might come in handy.

The next step is the housekeeping at 2000. Here we're going to find ourselves quite tongue-tied at the first run-through. We'll have some A$ answers and most of the CO$ movable objects, but every time we turn around in the line 3000 routine we'll find ourselves needing another A$ and perhaps even more CO$s for spice, so numbering the 2000 segment in increments of 20 is recommended. As the program progresses and you come up with new A$s, simply tack on each one after the last one here and go back and update the DIM in the first lines. ''Space Derelict!'' began with A$(1) through A$(7)—the rest were added as the situations arose. The same is true of the miscellaneous variables at the end of this section. It's almost impossible to know what these are before you create them, and you'll find yourself coming back and augmenting this list throughout the rest of the programming.

Keeping in mind the fluctuation of variables, now you can enter the replay and game-saving routines starting at 30000 in INTRO and CONWCOM, respectively. And remember: Every time you change a program variable anywhere else, *change it here too.*

Next you have the WINNER and LOSER routines, which aren't too hard to get right the first time. And then your last easy job—except that it's off-the-cuff programming that will require more work and thought than all the rest of the programming combined—is the miscellaneous business at 40000 in CONWCOM. Do your best and hope for the best, in that order.

Now we get to the crusher, the conversation segment at 3000 in CONWCOM. There's no way around this one, and the best thing to do is take it one word at a time. Try to think of every factor you can pertinent to that word and how it can change in every room, and then go on to the next word. No matter how much you do think of, you'll always leave out about half of it, so this is where line increments of 20 really come in handy. Using "Space Derelict!" as a guide should help quite a bit in covering a lot of the possibilities, but only in a general sense, as each new adventure is going to be radically different from every other. When you actually start running the program, you'll be amazed at all the mistakes you made, no matter how well you think you got it down the first time.

The final step in the programming is to make a printout of the entire adventure. This time the hard copy of the program is a must, because it is well nigh impossible to edit a program this long if you don't have it laid out on paper right in front of you. Also, the printout will let you carry the program and tinker with it anywhere you want—at work or at school or riding the train or walking the dog—all those places you can't take your computer. And now, with your printout in hand and a fully written adventure on your disk, you're ready to start debugging.

EDITING TECHNIQUES

The more complicated the program, the more problems there will be. For me there are two points in the creation of any new long program where I feel like throwing in the towel and starting something new. The first of these is in the middle of the actual entry of the program. By this time I know exactly where everything is going and I'm right in the middle of what seems to be endless typing. There is nothing creative about this part of the process, and I always wish I could hire somebody to do it for me. The second stumbling block is the debugging. You never know what you're going to find when you start unwrapping your program and trying to get it to run. Sometimes you'll find baffling problems where the program will seem to go off on some disastrous tangent that simply makes no sense. At other times you'll get syntax errors in lines that look completely correct, meaning you've made some mistake you didn't even know *was* a mistake. But one way or another you've got to solve these problems, and that's what we'll talk about now.

The first step in debugging is to take your printout and go over the program on paper. You won't find many logic errors at this stage, especially in an adventure where the logic is so simple. But you will spot some syntax errors, which are usually no more than typographical errors you made at the entry stage. My most common error is omitting the question mark when I type in PRINT statements (the computer, of course, translates the ? into a PRINT). So instead of having lines that read:

```
110 PRINT "YOUR NAME AND ADDRESS"
```
I'll end up with:
```
110 "YOUR NAME AND ADDRESS"
```
These are easy to spot and easy to fix. Punctuation is always a trouble spot, especially when you're concatenating with semicolons, as in:
```
666 INPUT "WHAT'S YOUR SIGN"; SIGN$
```
It's easy as pie to put the semicolon before the quotation mark. But this is the sort of thing you can pick up pretty easily simply by proofreading. My practice is to run through and mark all the changes on the printout, then LOAD the program and make the changes, checking off each change on the printout as I go. This way I still have an up-to-date printout, edited to match the actual program. I won't go into the EDIT functions of the TRS-80. The manuals explain these perfectly well, and experience will give you the hang of them.

The next step is to RUN the program and see what happens. With luck you might actually get into the first room, but I wouldn't count on it. First of all, the odds are that you'll encounter spelling errors or bad line breaks in the introduction that will have to be fixed. And more than likely the computer will spit back at least one syntax error at you. And if life is really against you, you might even branch into some no-man's-land you can't find any way out of. The first problems can be fixed easily enough, but crazy branches are another thing altogether. They require use of the TRON command.

TRON—TRace function ON—is the programmer's lifeline. No matter how complicated your program logic, TRON can follow it. All you have to do is track your TRONs and you're home free. But it's not always as easy as it sounds.

First of all, TRON can be used either as a direct command, as in:
```
TRON
```
or as a statement within a program, as in:
```
10 TRON
```
When you trace a program, the line number of every statement processed is sent to the screen. If a line contains two or more statements separated by colons, however, each statement will not get a repeat of the line number.

The key to the successful use of TRON is not to go overboard with it. When you use it as a direct command you end up tracing *everything,* and usually this is not necessary. A program will run correctly for a while and then go wacko. Find the last place in the program where everything was okay and insert the TRON there in the actual program. That way everything will run without the program lines where they are unnecessary (TRON can make a video display look like a number factory), and will start reporting those lines to you only on or near the trouble area. Keeping TRON contained like this makes it a lot easier to handle.

Your biggest use of TRON in an adventure will be in the conversation segment, but you may find yourself using it elsewhere as well. When all else fails, keep it in mind. By the way, TROFF is the command to deactivate TRON.

Once you've gotten past the introduction in your debugging, you'll find yourself standing at the threshold of your first room. The first command you should enter is SAVE GAME, to see if your saving routine is oiled and running. It's a good idea to do this as you enter every room. In a process like debugging where there are few time-savers, you should make the most of this one. When you save the game and everything is hunky-dory, the little red light will come on on your disk drive and without further ado will go off a couple of seconds later, and you will get a "Game Saved." followed by a new "What do you want to do?" prompt.

After saving a game, quit and start again. This time when you get to the choice of whether or not to play a saved game in the introduction, say yes and see what happens. You should find yourself back in room 1, period. There is a chance, however, that you'll first see some sort of TRSDOS error notice as the game-save file is being accessed. If so, there is probably a discrepancy between the number of items in the save routine and the number in the replay routine. Compare the two and fix the error.

Now you're back at the threshold of room 1, and SAVE GAME has been cleaned up. Next, make sure that all the exits are as they're supposed to be. Let's say you have a north exit and a south exit. First enter N. You should find yourself in the room to the north. Go back to room 1 and try S. This time you should move south. Now go back and try the incorrect W and E. These should *not* work and should bring to the monitor the appropriate no-can-do response (the same for U and D, if you're using them). Make sure this is all working before proceeding further.

Next, take a look at the room description. What you want to do now is make sure you've accounted for everything you've mentioned. Try your LOOK command for each item. If there's a DEAD DOORNAIL up there, you want to make sure your program responds when you say LOOK DOORNAIL. If this gets you an "I don't know what you mean by DOORNAIL" response, you've got to fix it. That's why we left increments of 20 in the 3000 conversation routine, to fit in all these sorts of additions that are sure to crop up. GOs should work the same way; if there's a CHAIR, the command GO CHAIR should get a meaningful response.

Next, try to take the movable objects. Pick up each one and see what happens and put it down again, performing all the variations on this theme that you can think of. If these don't work, fix them. Again, mark all the changes on the printout as well as making the changes in the program.

The final things to do are everything else. Start at the one-letter-word commands and run up through the nine-letter ones, trying both the ones you've included as well as ones you haven't. Make sure the responses are appropriate for each situation and change them if they're not. Work on both your nouns and your verbs, because there could be all sorts of bugs just waiting to jump out at you. And when you've finished all that, go into the next room and do it again.

As you progress in this fashion, you'll find fewer and fewer bugs, but there will probably be one or two rough edges that can be ironed out in each room. But eventually you'll catch them all and have the semblance of a finished game. When you think you've got them all, run through the game doing all the right things, solving all your complications, and see what happens. You should win in about five minutes. If you can't win your own game, how is anybody else going to do it? When you're satisfied that everything is working properly, you might want to renumber it all for the sake of neatness and print another listing of the (almost) finished program for your records. Of course, these steps are entirely optional.

The next step in the procedure is highly recommended, but it's a two-edged sword that requires caution. Give the program to an adventure-playing friend and ask him to play it on his own for a while (not under your prying, watchful, snoopy eyes). Tell him to call you if he gets any breaks in the program (he shouldn't get them at this point, but he probably will) or anything else that doesn't gel. But don't have him call you for help in playing the game; that's his problem. The nice thing about having somebody else help with the debugging is that he can find things that you, as the designer with complete knowledge of the inner tickings of the program, might easily overlook. The bad thing, and what makes it a double-edged sword, is that this process is not unlike giving a friend a copy of your great American novel. If your game's a complete stinker, do you really want your friend to tell you that? Will you really accept his opinion that it's too hard or too easy? Will he have good ideas to improve the play of the game, better complications, whatever? And if so, how will you respond? I can't help you out on this, because I have the same problem myself. Designing a game—or any program—is a creative process in which in one respect or another you put your ego on the line for others to tear apart as they see fit. Even abject flattery from your players will do nothing to relieve the fact that you will probably never be completely satisfied with your game. In that regard there is no difference between creating a game or writing a story or making a film or any other imaginative activity. All I can do is quote two truisms. If your work does not succeed, find out where the problem is and work at getting it better. And if your work does seem to succeed, keep at it and improve it until it succeeds even better.

6
TIDBITS AND TRIVIA

In ''Space Derelict!'' you've seen a lot of programming doing a lot of things and gotten a complete picture of one way of creating computer adventure games. I do want to reiterate a couple of points that are important to keep in mind, especially if you are a relative newcomer to programming. The first of these is that what you have just seen is not the only way of doing it. For my money it's a good, simple way, but it's certainly not the only way. Even as we went through the program I pointed out areas where a tightening of the program statements could make for more elegance, or where perhaps a section could be relocated to enhance processing speed a bit. It is important to remember that this adventure was designed as an instructional tool; if one were so inclined, one could take the exact same game and design it in such a way that it would run identically but so that nobody but the programmer could understand it—which would have had you demanding your money back on this book if I had done so myself. Following to the letter everything I've said here will give you perfectly good adventures; improving on what I've said will give you perfectly better adventures. You have the map now; where you travel is up to you.

Another point that must be made is that this program design is not particularly orthodox in its methodology. The nice thing is, it doesn't have to be, but if you stop reading this book now I don't want you to think that every program can be designed the way this one has been. There is a concept kicked around in computer circles called ''structured programming,'' which has nothing to do with anything we've been discussing so far. In essence, structured programming is a step-by-step process of design where a program is constructed as a series of independent blocks, each block consisting of a particular subroutine to do one small part of the program's whole. For the most part, that is the way programs ought to be designed, and in fact that is the way our poker game is laid out, so we'll be going into programming theory in some detail in Part Two of this book. But by pointing this out now I feel I've done my duty and silenced in advance those structuralists who will decry me on the basis of just this one program. My credo, which I've made no bones about all along, is that if it works, it works. Although there are some definite rights and wrongs that must be addressed in programming, there is no point in getting hung up in certain academic stylistics just for the sake of those stylistics. If I ever write my book *Zen and the Art of Personal Computing* I'll go into this in more detail, but for now let's resume our regularly scheduled program, which is already in progress. The contents of this chapter are some odds and ends I didn't work into "Space Derelict!" They're not drawn out in glorious Technicolor, full-screen detail, but they'll get you started on just about any other programming situation you might encounter in an adventure.

AN ELEMENTARY ADVENTURING CLOCK

One thing I hate about TRSDOS is having to enter today's date before I can run it. I do my computing in a room without a calendar, so I always have to make up a date. Unfortunately, I do usually know the day of the week, which never seems to be the same as the one TRSDOS delivers for my imaginary date. Oh, well . . .

A corollary of date monitoring is this: If you want, you can have TRSDOS tell you what time it is. After putting in today's (or some day's) date, put in the time:

```
TIME XX:YY:ZZ
```

where X is hours, Y is minutes, Z is seconds. Allowing for the fact that some TRSDOS commands momentarily disable the timekeeping, you'll now have a semi-accurate $2,000 clock in front of you. In fact, type

```
TIME (CLOCK=YES)
```

and the time continuously displays at the top right corner of the screen (change the YES to a NO to get rid of it). Once you're in Basic, the way to find out the time is to

```
PRINT TIME$
```

TIME$ is the function that can call up the time from within a program. If you haven't specified any TIME for TRSDOS, the computer nonetheless still keeps time for you, beginning with 00:00:00 when you boot TRSDOS. This means that you can actually use values of TIME$ for real-time purposes in your Basic programs. Very nice indeed, but they are no help to us as adventure designers.

It is not inconceivable that you might want to incorporate a time factor into an adventure. A bomb might be scheduled to explode at midnight, perhaps a treasure must be discovered before the moon is in its fourth phase—there are all sorts of possibilities along these lines. The problem then becomes figuring out some way to measure the imaginary time taken by the player in each of his moves. The simple part of this is the establishment of the deadline—such and such an action must take place by such and such a time, or else. The hard part is coming up with the measurements.

The most elementary and probably the least satisfying approach to artificial timing is to limit the number of player moves. You might throw in a new variable, PM (for Player Move), and augment this variable every time the player comes around to the beginning of the 3000 conversation segment:

```
3005 PM = PM + 1: IF PM = 100 THEN GOTO losing scenario
```

There's not much subtlety to this approach. It simply hits up PM on every run-through until it equals 100 (or whatever number you decide), and then the program branches off to the losing routine (which should contain a special proviso to advise the player that the reason he has lost is that he spent too much time). This can be augmented a bit by a warning to the player:

```
3005 PM = PM + 1: IF PM > 90 THEN PRINT "YOU ONLY HAVE ";
100 - PM;" MOVES LEFT.": IF PM = 100 THEN losing scenario
```

This at least gives the player some knowledge of where he stands and why.

A nicer way to handle this, which is not all that much more complicated, is to give a starting time and a finishing time and break down the difference into recognizable segments. Let's say your scenario starts at noon and the deadline is midnight. That gives the player twelve hours. Let's divide each hour into 12 moves, which means that the player has 144 moves in which to win the game. After every 12 moves you could tell the player that another hour has just passed and give him a new reading on the time:

```
3005 PM = PM + 1: IF PM/12 = CINT (PM/12) THEN HR = HR + 1:
PRINT "IT IS NOW ";HR;" O'CLOCK.": PRINT "YOU HAVE ";12 -
HR;" HOURS LEFT.": IF HR = 12 THEN GOTO losing scenario
```

This is how we would advise the player that an hour has passed. Each time we come to the beginning of the 3000 segment we again augment PM. The second statement on the line (IF PM/12, etc.) handles the hourly divisions. CINT(PM/12) is, of course, a whole number, an integer—that is the definition of the CINT command. What we're saying here is that if PM divided by 12 equals a whole number, then this means that one hour has passed, and the HR (hour, of course) variable is augmented accordingly. When PM reaches 144, HR will equal 12, which means that the player will see the line:

```
YOU HAVE 0 HOURS LEFT
```

and will immediately find himself facing your losing scenario.

This can, if you are so inclined, be stretched into days. Let's say we're starting at midnight and we want to give the player the next three days. In that case we would simply plug in:

```
IF HR = 24 THEN DAY = DAY + 1: HR = 0: IF DAY = 4 THEN losing
routine
```

This follows the same logic as before, merely extending it by dragging in a new variable. A further elaboration could allow the player to input the command TIME or TELL TIME. We would respond in the conversation segment:

```
IF C$ = "TELL TIME" THEN PRINT "IT IS DAY NUMBER ";DAY:
PRINT "THE TIME IS NOW ";HR; "O'CLOCK."
```

You could make this even cuter if you wanted by including PM as well and giving hours and minutes. We're starting to get lengthy here, so I'll let you contend with that bit of business yourself.

The next step in making this even more sophisticated is not to penalize the player for moves that don't actually happen. Let's say the player enters the garbage command TAKE CRONW instead of the correct TAKE CROWN. Should he have to pay the price for this? I say no. But this requires a bit of work. It means you have to go through the conversation segment catching each and every instance where a player input is kicked back rather than acted on. Simply adding the line GOSUB 49000 right before the GOTO 3000 in these cowcatcher lines will send you off to:

```
49000 PM = PM - 1: RETURN
```

With this step the incrementation of PM that occurs in 3005 will actually be resetting PM back to what it was before the last bad move was made.

Finally, there is the bend-over-backward version of all of this recommended only for the most masochistic programmers. When you think about it, every action taken by the player does not in fact require the same amount of imaginary time. Going north through an open door is a much quicker process than defusing a 50-megaton nuclear bomb, for instance. Swimming north through a swamp takes longer than running north on a road, and so forth. Most of your player activities will be roughly similar in time elapsed, but some are different from others and can be handled with a nod to these differences. Are you ready to go through your conversation segment and augment PM by different values for every different activity, with a default value of 1 to be handled as before at 3005? What a nightmare—but it could be done if you were willing.

As with anything else in your adventures, if you're going to put in a clock, put it to good use. Simply limiting player moves has its point, but as long as you're ticking away the hours, why not make the most of it? You could have a clock go off (the appropriate number of SOUND commands) whenever the hour is struck. You could have ghosts pop out only at midnight. Or werewolves pop out at the hour of the wolf, 4:00 AM. You could have lunch appear on the table at noon, dinner at eight, tea service at four. As with everything else in adventures, the possibilities are endless if this particular sort of complication tickles your fancy.

VERIFYING THE SECOND WORD

It is possible to include in your adventure program a definitive routine that would analyze the second word of the player input and verify whether that word was one your program recognizes. This is not very hard to do, but it does require that you comb through your game to pull out every possible second word so the analysis can be made. If you miss even one word, the whole thing is for naught. So it's time-consuming and nit-picky and very much optional (which is why I omitted it in ''Space Derelict!''), but it is a nice piece of polish you might want to incorporate.

To pull off this hat trick we're going to use READ . . . DATA statements. If you know what these are all about you can skip down a few paragraphs, but if you're like me you probably never found much use for these oddities (or maybe you don't understand them at all), so bear with me for a while.

Like a disk file, DATA is a storage container, only within a program rather than onto a disk. And READ performs the function of reading through the DATA from start to finish (unless some other modifying parameter is set), doing nothing whatever other than making the data live within your program for other statements in the program to operate on. The problem is, it's very hard to see much purpose to this operation in the abstract.

Here's an example of READ . . . DATA:

```
10 FOR X = 1 TO 5: READ D$: PRINT D$: NEXT
1034 DATA 1,97,43,16,HIKE
```

You're probably somewhat aware of the basic working of READ . . . DATA statements, but we'll restate it here. Whenever a READ statement is encountered in a program, the computer runs down to DATA and READs off a piece of it. The first READ reads the first piece, the second READ the second piece, and so on. If there's so much data that you need more than one DATA statement, READ simply proceeds from the first to the second to the third, in order, reading each piece as required. So in the example above, running the program would give us the following results on our monitor:

```
1
97
43
16
HIKE
```

A change of line 1034 to:

```
1034 DATA 1,97,43
1035 DATA 16, HIKE
```

would give us the exact same results. Since we are calling these pieces up as strings (D$), no quotes are used and the numbers are treated as a string just as the HIKE is. If we had used all numbers, a simple READ D would have sufficed.

In the example given, when we get to the end of line 10 (that is, when we hit the fifth NEXT), we have exhausted the DATA storage. If we're finished with it, that's fine, but if we need it again, we could add:

```
20 RESTORE
```

so that the pointer that was making sure we took each piece in order will now be set back at the first piece, so that a subsequent READ statement would start at the beginning again (with the number 1 in line 1034).

Of course, we don't always have to READ all the pieces in a DATA statement if we don't want to.

```
10 PRINT "PICK A NUMBER FROM 0 TO 9"
20 INPUT X
30 READ Z
40 IF X < > Z THEN PRINT "YOU LOSE!": GOTO 10
50 PRINT "YOU WIN!": RESTORE: GOTO 10
60 DATA 3,0,9,4,1,7,2,2,4
```

Here's an extra game thrown in at no extra cost, and if it's dull as dishwater, at least it demonstrates READ . . . DATA. The way this game will proceed is simple. The player will be prompted for a number from 0 to 9. After inputting the number the program READs the first DATA element, in this case the number 3. If the player has guessed incorrectly, we go back to line 10 for another try. This time the correct answer is 0. Wrong again? Back to 10, the correct answer is 9, and so forth. But if the player guesses correctly at any point, then the RESTORE statement in line 50 puts the pointer back at the first element, the number 3, and the whole thing starts over again.

Unfortunately, there is a flaw in this extra give-away-for-nothing game. What happens if a player makes nine incorrect guesses? At the tenth run-through there are no more pieces of data to be read, so I'm afraid the program will crash as a result. But there's a way around this, too.

```
5 IF Z = 10 THEN RESTORE
10 PRINT "PICK A NUMBER FROM 0 TO 9"
20 INPUT X
30 READ Z
40 IF X < > Z THEN PRINT "YOU LOSE!"
50 IF X = Z THEN PRINT "YOU WIN!": RESTORE
55 GOTO 5
60 DATA 3,0,9,4,1,7,2,2,4,10
```

Now we've covered our flanks 100 percent even though we've thrown in a ringer. We've asked our player for a one digit input, so if he's not cheating there's no way for him to input the number 10, so that extra number at the end of our DATA statement is a signal to the program that the DATA have run out; line 5 will handle that situation for us by making sure we never run out of pieces.

However, none of this really gives you a good idea of what else READ . . . DATA statements are used for. The most common use I've seen is for large-scale POKEing of numbers into a program either for graphics or machine language arcana. But we're not going to go into any of that business here, so where does all this leave us adventure designers?

There is a very easy way of taking the player input right off the top and making sure that the noun—the second word of the input—is one we have covered in the conversation segment. The first thing we have to do is break up C$ into the two elements S$ and S2$. If we have no S2$ (i.e., C$ is a one-word input), everything would continue as normal. If we do have an S2$ we would then branch off to somewhere like line 49000 and try this:

```
49000 REM SECOND WORD CHECKING SUBROUTINE
49010 READ SW$
49020 IF SW$ = "FINIS" THEN PRINT "I don't know what" + S2$ +
"is": RESTORE: RETURN
49030 IF MID$(52$,2) = SW$ THEN RESTORE: RETURN
49040 GOTO 49010
49050 DATA DOOR, HOLOGRAM, NORTH, SOUTH, PANEL, CEIL-
ING, DEVICE, et cetera, et cetera, et cetera,FINIS
```

This would do the job very neatly. If we get a match in line 49030, then the subroutine is terminated, the DATA restored, and the chips will fall where they may in the remainder of the conversation with the computer segment. If we get to the end, which is FINIS—the last piece of data—then we print the message that the word is a dud and go back whence we came. Keep in mind that a cowcatcher is now necessary immediately following the line that got us here in the first place (the GOSUB 49000) to the effect IF SW$ = "FINIS" THEN 3000, which would get us back to the beginning of the conversation segment.

CHANGING A GAME IN PROGRESS

Everything we've seen so far has given us innumerable possibilities for game design, but there has been one limitation that we only broached briefly in our discussion of miscellaneous business. Up till now the actual adventure has been static; although the player moves around in it however he may choose, at his own pace and his own inclination, the game itself remains the same. Unless he takes movable object X from room Y, movable object X will always remain in room Y. The game is entirely at the command of the player, entirely lacking a kinetic energy of its own. The player just goes along and the game universe just sits there as he roams around in it. There is nothing wrong with this per se, and some very interesting and classic adventures have had this quality about them. But there is no reason why we can't expand our game universe and give it a bit of life on its own, and that's what we'll be talking about now.

Right off the bat, I don't want you to think I am proposing the introduction of random factors into an adventure. Quite the contrary: As far as I'm concerned, anything random in an adventure goes counter to the whole point of what an adventure is all about (although one might make an exception for some minor randomizations—the ability of a player's gun to hit a specific target, that sort of thing). An adventure should be a well-thought-out puzzle with dozens of twists and turns to challenge the player's intellect. Once things start becoming random the challenge is less on the player's intellect and more on his hunches or his reflexes—fine for an arcade game but not the stuff that adventures are made of.

The kinds of changes I have in mind here are in the nature of the normal comings and goings of people and things that would normally transpire in a real universe similar to the one in your game. In the basic module they belong in the line 40000 miscellaneous-business area. You can make these incredibly complex or quite simple, depending on how deeply you want to dive into it. Say, for instance, that you have a snake in your game, out to attack our player. Of course, you could just keep the little viper waiting in a given room, ready to pounce, but being an amateur herpetologist you feel your snake should laze in the sunshine during the day and laze in his den during the night. No problem. You build in a clock as we discussed before and decide which hours you want to be sunlight and which darkness. Then you change the CO(X) value of your snake to reflect the appropriate sunny or dark room wherein the snake is lurking. This is a simple enough process and is easily adaptable to a number of circumstances.

The example above requires a clock, but you might want a different sort of action dependent not on independent factors but on the player's moves. For instance, you might have in your game a vampire who is stalking the player. The vampire could move in a random pattern from room to room (this is another example of reasonable randomness), but wouldn't it be much more clever to have him able to sense where the player is in order to track him down? Again, no problem, but this time a lot of work. The requirement here is a room pattern laid out so that whenever your program knows both the R value of the player and the CO(X) of the vampire, the program is able to decide which adjacent room to move the vampire into in order to put it that much closer to the player. No mean feat, that. It requires one of two things: a room layout such that a simple mathematical formula will always quantify the differences between two rooms and provide the next closest room from one to the other, or else a very dull programming operation where every room is analyzed in relation to every other room, giving the closest in each case. The first option is infinitely more desirable but requires much work at the conceptual stage. The second option simply requires a lot of dull computer hacking.

There are other ways to move things around, however. You could base a movement on the fulfillment by the player of some special criterion. For example, let's say that if the player enters the restaurant through the front doorway, the sinister dishwasher exits through the back one. This is simple enough to handle in the GOs. Or you could give a game character an entire itinerary to follow and simply have that character move in a prescribed fashion from room to room with the completion of each player input. Again, this would fall into the line 40000 miscellaneous-business section and would require a complex loop monitored by a special variable:

```
41320 REM KGB PETE MOVEMENTS
41330 KG = KG + 1
41340 IF KG > whatever the limit is to your loop THEN KG = 0:
GOTO 41330: REM START LOOP AGAIN
41350 ON KG GOTO a number of different places immediately
described below starting with 41360
41360 CO(99) = 101: GOTO exit from routine: REM CO$(99) = ''KGB
PETE''
41370 CO(99) = 102: GOTO exit
41380 and so forth and so on until
41490 REM EXIT FROM KGB PETE ROUTINE
41500 IF CO(99) < > R+100 THEN RETURN
41510 GOSUB 13000: REM DISPLAY PETE ON SCREEN
41520 RETURN
```

These lines above are fairly straightforward and understandable with all we've covered so far, except for the ON statement. Line 41500 covers the condition that KGB Pete and your player are not in the same room. If that is the case, the display of movable objects in the room need not reflect Pete's presence, so you can return from the subroutine without a care. But if Pete *is* in the same room, you want to make his presence known, which means making a trip to the 13000 room inventory routine. The only problem is the ON statement.

ON is one of the most ingenious commands in the Basic vocabulary and can be used either with GOTO or GOSUB. As in the example, ON looks like this:

```
ON X GOTO 100, 200, 300, 400
```

or

```
ON X GOSUB 100, 200, 300, 400
```

ON reads the value of X (or whatever variable follows it), then uses that value as a counter to know where to GOTO or GOSUB as the line dictates. If X=1 then we'll branch to line 100, if X=2 then 200, if X=3 then 300, if X=4 then 400. The ON X counts the possibilities, picks the right one, and off it goes. If there aren't enough possibilities, say X=5, or X=0 or a negative number, processing will simply pass to the next line in the program. We'll be using ON quite a bit in the next part of the book.

We have now covered in this section a few possibilities for internal movement within your universe. As you can see, none of these things is a simple one-liner. Although for the most part the programming is relatively easy, it can be long and time-consuming. So use these ideas wisely and frugally. They can liven up a game, but they can also deaden up a game designer.

NOTES ON A D&D SCENARIO

I would find it very surprising if anyone reading this book didn't know what D&D—"Dungeons and Dragons"—is all about. So many adventure games are direct derivations of the D&D phenomenon that I simply can't imagine a computer hacker being unaware of so seminal an object. Only a few years ago you had to seek out D&D literature far and wide; now it's available in almost any bookstore. But for those who have never played, one brief paragraph before we get down to business.

Like a computer adventure, "Dungeons and Dragons" (and its various offshoots—the original is very much protected by trademark by a company called TSR Hobbies, Inc.) is a role-playing game. An artificial universe is created for the players to roam around in, on quests either for treasure or to perform specific tasks. Players take on the characteristics of warriors or wizards, thieves or clerics, whatever suits their fancy, human or otherwise. As time passes, a player character gains experience, and with that experience comes increased power and knowledge. The rules for D&D are not etched in stone (although there is much guideline literature), nor is the game universe a fixed, given commodity (although some prefabricated universes are available in the stores). The structure of the game rests with the player chosen as Dungeon Master. This player designs the universe, peoples it with nonplayer characters, stocks it with treasures, spices it up with traps. This player moderates the moves of the player characters, arbitrates what can and cannot happen within that universe, and consults the manuals when the manuals need consulting. Not unlike an adventure game designer, eh?

The big problem with getting a good D&D game going is that you need a fairly large group of dedicated players. A single game can take months, while a player character can last through dozens of games. A band of dedicated comrades-in-arms is hard to find outside of school, the army, or very weird job environments, and those are where most successful games are conducted. But for solo operators, the computer provides a close second, via the adventure game.

The role-playing in a computer game is intrinsically related to the role-playing in D&D, but there are some elements of this sort of game that we have not covered yet and that we should touch upon briefly. The biggest difference is the element of combat, which we've only barely handled in our attack of the killer robot in room 23 of "Space Derelict!" But let's say you want dozens of killer thises and thats roaming your hallways, and more, you want your player to be able to kill them not only by wits but also by brute strength, if such strength is called for in your scenario. This requires that both your player and your killers be imbued with quantifiable attributes that can be measured against each other, with a slight random agent thrown in for good measure. Let's say you boil it down to three variables: ST for strength, SP for speed and reflexes, and WI for wiliness and wit. An average man would have average strength, average speed, a bit better than average wiliness. A superman type would have the same wiliness, most likely, but greatly increased strength and speed. A dwarf would have greater-than-average strength, less-than-average speed, and the same wiliness as the average man. A wizard would have less speed and strength than the average man but much greater wit. A tiger would have strength and speed and no wit, a snake speed and wit but no strength, a furgle-eyed difflesnopper would have whatever you'd want to give it.

Each combatant would have a number applied to each of his three variables—from 1 to 20, say—and this would begin to account for his possibilities in combat against another combatant. It is up to you as designer to decide just what beats what in the actual arena. Will an SP of 9, an ST of 15, and a WI of 4 beat an SP of 2, an ST of 6, and a WI of 19? Who knows? You must include in your game a formula for determining just what beats what. Such a formula is known in programming as an algorithm. Algorithms make the computer world go round. The algorithms that guide the activity of a computer in a given situation—where certain seemingly unquantifiable concepts are indeed quantified, analyzed, and acted upon—are the real crux of what program design is all about, and we'll go into them further when we talk about designing strategy games. For now I'll suggest only how the SP, ST, and WI algorithms might work within an actual program. A player would encounter an enemy and first be given the option to run or fight. If he runs, the enemy gets a free shot at his back, but with only slim chances of getting a hit. If a hit is made, then the player must turn and face combat; if he's not hit, he's home free for the time being. If combat is engaged, you either have to set it up as a series of hits and measure the success or failure of each hit with your combat algorithm, or else your algorithm will simply say IT'S DEAD or YOU'RE DEAD. The former approach has a bit more style to it.

None of these combat variables should remain static. I would imagine one's strength, for instance, would be the measure of one's existence. If your enemy keeps hitting you, you lose more strength until your strength is zero, and it seems to me that that would make you a goner. But a 97-pound-weakling character ought to be able to gird his loins and get himself some weaponry to up his strength a bit. A player with a 10 in ST could have that increased to 15, say, if he's got a mace tied to his armor (or a can of Mace in his pocket—who knows what kind of game you're designing here? For the record, I use the masculine pronouns throughout this book both for the sake of convenience and because, let's face it, this kind of game-playing does tend to be more a male than a female diversion). Of course, once you start getting into weaponry you most likely have to create a whole arsenal, with each individual item not only accounted for as a CO(X), but also with its ST thrown in for good measure. Then there's the question of magic weapons that have special powers—perhaps secret, perhaps not. There are magic spells available that would certainly up anybody on the WI scale. When you start getting into this you can begin to see why a good D&D-

type game is hard to find. Not only do you have all the elements of your basic adventure, but also a whole new batch ultimately just as complicated thrown in on top of them. My suggestion would be to ease into this sort of thing slowly after you're fairly confident about manipulating the basic adventure module. Don't start by throwing in 100 different monsters of all shapes and varieties; two or three will be good enough, with a simple, boiled-down algorithm to guide them, and as you watch them shake down in actual play you'll begin to get a feel for how they could be improved and how your algorithm could be expanded to make the combat feature that much better.

There is another idea I've had on D&D for a while, and I even know one person who tried it but ultimately abandoned the idea when he realized just how much chewing he had facing him after he took that first bite. Being a Dungeon Master is not easy and requires a lot of looking up data, throwing dice, that sort of thing. Why not write a program for Dungeon Masters that does all of the dogbody work for them? They'd still have to arbitrate the squeakers and orchestrate the universe, but they wouldn't constantly be figuring out this one's hit points vs. that one's strength, with the plus-two sword factor and all that sort of stuff. I wouldn't be surprised if such an item is already on the market, although I've never seen it, but if you're an experienced enough Dungeon Master you wouldn't need much programming savvy for what would basically be a simple sorting-and-storage program.

KEEPING PLAYERS OUT!

Here's the problem. You've written this wonderful adventure that's taken you months and months to perfect, and some clown comes along and breaks right into it and starts taking the program apart bit by bit to find the solutions not by playing it but unscrewing it. The sort of copy-protection that accompanies some of the software you buy in the stores is certainly economically unfeasible to amateurs (and, quite frankly, akin to shooting gnats with bazookas when applied to homegrown products). And let's face it, if the determined software pirate can break into disks that the pros have protected, there's not much you can do against the Sunday-morning pirate who wants to break into yours. Nonetheless, there are tricks built into TRSDOS that you can employ that will prove quite frustrating to anyone trying to get into your programs. It all boils down to passwords.

After you've finished writing and debugging all the programs on your adventure disk, think up a short password that you won't forget—going through life with but one password is a good idea. There's nothing worse than being denied access to your own program. Now, password firmly in mind, enter the SYSTEM command and then the following:

```
ATTRIB INTRO (USER=,OWNER=MOM,PROT=EXEC)
```

ATTRIB is the command that controls and allocates your password operations. What we're doing here is ATTRIButing to the INTRO program the operations in the parentheses. USER= with nothing after the equal sign means that anyone walking past your computer can get something (as yet to be determined) out of INTRO without a password. OWNER=MOM sets

MOM as the password that gives you, the so-called owner, unlimited access to INTRO. PROTection = EXECute means that the only thing the passwordless user can do is execute—i.e., run—the program. Try it. Run INTRO and see what happens; it should run fine. But try to list it—illegal function call, the machine says. You can run it or load it, but you can't list it. Try it as LOAD "INTRO.MOM", however, with the name of the program followed by a period and the password, and you can list it, or anything else, till the cows come home.

There are other shadings to ATTRIB that you might find valuable. Look it up in the manual. For us, denying our player any access to our program except to run it, while we maintain full control, seems more than enough.

PART 2

STRATEGY GAMES

7

THE ART OF STRATEGY

GAMES, STRATEGY, AND ASSORTED BALONEY

I debated with myself for quite some time whether to burden this book with all my theories about games and game-playing. There is much to be said on the subject, and much has been said and continues to be said almost daily by various experts and pseudoexperts. The question of what games are and why they are played seems to be the concern of just about every branch of both the physical and the social sciences. Anthropologists wonder about game-playing among chimpanzees, psychologists try to make sense of children playing kick the can, sociologists equate business management with game-playing, and there is even a branch of mathematics devoted to game theory. Admittedly all these people tend to have different purposes in their studies, but nonetheless there is much overlap in what they're studying if not exactly in the way they're studying it. The question then is, why? Why have all these serious Ph.D. sorts taken it to themselves to try to figure out what the apparent frivolity of gaming is all about? In a way it depends on which Ph.D. you happen to be talking about, but one thing is certain: Games are a major facet of human (and perhaps in some cases nonhuman) existence. People play games; apes play games; cats and dogs play games. And it's curious that now that we've created artificial intelligence—computers—one of the first things we've done is teach our computers to play games as well. Score another point for all those Ph.D.s.

As I said, I have a lot of theories about games, but I really don't have all that much to back them up, and this is not the place to go into them anyhow. But now that we are in a position to make game players out of computers, we ought to have some idea of what games are all about so we'll be able to go about our game-designing in the best way possible. And the first thing we have to do is disabuse ourselves of the idea that adventures such as ''Space Derelict!'' are true games. They are not. They are a form of puzzle, like a crossword or a 5000-piece jigsaw or a mathematical brain-teaser. They are an entertainment, they are a pastime, but they are not a game. They have simply acquired the game sobriquet by default because they're like games and therefore can be roughly lumped together into the game classification. But we are too clever for that by half and will not make a similar mistake.

By the same token, we must exclude arcade-type games from our discussion here. Shooting aliens out of the sky, collecting hostages in your chopper, blasting enemy submarines—these are all a lot of fun and share some fairly important characteristics with true game-type games, but they primarily fall under the heading of games of technical skill, where the player's success or failure rests on his ability to manipulate items physically, in this case computer blips. The game of pool would fall into a similar category; one does employ game-like techniques against one's opponents, but ultimately one's success or failure rests on one's ability with a cue stick, a very technical skill. Pool certainly requires greater technical skill than ''Galaxians,'' but I'm sure you'll agree that the analogy is apt.

Trying to define games is like trying to define man. No matter how fine a point you put on it, sooner or later someone will come along and knock you down. Man is not the only creature to walk on his hind legs, not the only creature who thinks, not the only creature who makes tools, not the only creature who can communicate symbolically, and so forth and so on.

The same difficulty of definition applies to games. I would like to say that a game is a structured cerebral conflict conducted according to a series of set rules by a certain number of players, eventually resulting in clearly defined winners and losers, but before I could get the words out of my mouth you could cite right off the top of your head a dozen perfectly valid games that do not fit that description. But it really isn't that bad a definition, and it certainly does cover the kinds of games we'll be discussing from this point on, so let's go with it.

For our purposes, the most important part of a game is, if you will, its soul. A good definition of the word game would include both tic-tac-toe and chess—both have the conflict, the rules, the winners and losers. But there is a world of difference in the complexity of these two games. The souls of these games are entirely different.

Let's look at tic-tac-toe for a second. This is a game of very simple rules and a very clear objective, connecting three marks in a row. Because the game is so simple, its soul is almost as obvious as its rules. Even most children light on the winning strategy for tic-tac-toe very quickly, which leads me to my axiom that in fact it is a game's strategy that is its soul.

If both players can connect with the soul—or strategy—of tic-tac-toe, this is what will happen. The player with the first move will take the middle square. The player with the second move will take one of the corners. From that point on, short of an incredible blunder, the game will always result in a draw. Now, if only one of the two players understands tic-tac-toe strategy—if only one player is in touch with the soul of the game—that player stands a good chance of winning quite often. Not every time—because there are so few possible moves it is easy for the ignorant player to stumble onto a good one—but often enough. If neither player knows the strategy, it's anybody's game. And again, if both players know, short of a blunder it's always a draw.

You may not buy this, but all the above applies as well to chess, with only a difference of degree. Chess is an enormously complex game that no one has ever totally mastered. Some players have been phenomenally good, but none has ever been fully in touch with the soul of the game. So we take two players with equal (but not perfect) understanding of strategy, and what do we get? Well, we don't draw every time, that's for sure, because the possibilities for minute, subtle blunders are rife. But over the long term, playing both the black and white pieces, two equally matched players will roughly break even, each winning and losing about the same number of games (plus a lot of draws). With unevenly matched players, we find that as the difference in skill increases, the game more frequently goes to the better player until at some point the lesser player doesn't have even the slightest chance of a draw, much less a win. Chess allows for subtler degrees of skill than tic-tac-toe, but eventually it all boils down to the same thing: The player who has the better strategy wins; the player who does not, loses.

This is as good a time as any to differentiate between strategy and tactics. Both are important, but they are not the same thing, and as a prospective game expert you should be able to distinguish between the two. Strategy can be loosely defined as your overall plan for obtaining your major objectives. In chess, good strategy includes getting as much control as possible of the center of the board. Another part of good strategy is getting all your pieces out early in a strong offensive position while maintaining a solid defense as well. Other facets of your strategy might change as the game develops; white might go for a king-side attack while black goes for the queen side in some Sicilian variations, for instance. In any case, strategy is your grand scheme, and optimally your strategy is multifaceted and capable of adapting itself to new situations as a game progresses. Strategy is not to be confused with your goal, which is the capture of your opponent's king. That is your objective; your strategy is your plan for obtaining that objective.

Your tactics, on the other hand, are concerned with the immediate question of the conduct of the battle. You might be thinking, I want to control the center of the board—your strategy. But you have to decide how to do it, perhaps a vanguard of pawns flanked by both bishops. That is tactics. In World War II the Allied strategy in 1944 was to establish a beachhead in Europe through which to supply fighting troops to knock out Germany's thinly drawn Western defenses. The tactics employed to meet this strategy were the use of all those LSTs in Normandy, the parachuting of John Wayne behind enemy lines in *The Longest Day,* the blowing of this bridge, the blocking of that road, etc.

What does this all mean in terms of programming computer games? I want to communicate a feeling for the concept of strategy, because I will be throwing that word around quite a bit in the discussion to follow. Every game has a soul, a strategy. Some games are so simple that this soul is easily accessible, and therefore just about any player can achieve the perfect strategy. As the complexity of a game increases, it becomes harder and harder to find that perfect strategy, but it is still there somewhere. Many players can do little more than try to choose decent tactics in the heat of the moment—this describes the vast majority of chess players. But within every game, somewhere, somehow, that soul, that perfect strategy exists. And as a game programmer it is your responsibility to get as close to that soul as possible.

One thing that should be obvious from all this is that it would be well nigh impossible to program a game with which you are not completely intimate. If you don't completely understand the strategy and tactics of a given game, there is no way you can get the computer to do what you can't do. And by understanding a game I mean that you know it inside and out. You know the perfect strategy, you know less-than-perfect strategies, you know the good tactics and the less-than-good tactics. I mean that you understand the game at every possible level. I do not necessarily mean that every time you play this game you always win, nor do I mean that you intend to program the computer always to win. But you have to be capable theoretically of both for one very simple reason: If you don't know your game well enough to do that, your program won't be very good.

ALGORITHMS—INTEGRATING GAMES INTO PROGRAMS

The soul of a computer program is the algorithm, a term we promised to explain in detail when we reached this part of the book. So what, pray tell, is this mean-sounding animal? Putting computers aside for a moment, an algorithm is simply a formula that quantifies data in a meaningful way so we can act upon those data.

The human mind is capable of creating algorithms nonstop for every situation we encounter in life, and we never think about how this process is conducted. Somehow our minds are capable of quantifying just about everything we put into them, and one way or another giving us an answer, a choice, a course of action. For instance, I accidentally touch a red-hot frying pan. I now have innumerable options open to me. I could remove my hand from the pan, I could scream out in pain, I could hop back and forth from one foot to the other, I could sneeze, I could blow a raspberry. But my brain very quickly eliminates the supercilious responses, such as the raspberry and the sneeze. On a scale of 0 to 100, my brain instantly rates those two actions, and all others like them, a 0. Hopping back and forth, which I may indeed do shortly, is still rather nonessential, so the brain gives it about a 4 and holds onto it for further action later. But screaming out in pain and removing my hands are both involuntary reflexes, and both will get very high ratings. Screaming will get a solid 99, and you can be quite sure that my brain will have kept this high on the stack of responses and will get to it shortly. Moving my hand away will, of course, get a 100 rating, and that is the first thing I will do. And believe me, I will do it very quickly.

The point of this somewhat shaky analogy is to show how related data can be quantified by a processor, even though those data do not come attached with quantities clearly labeled. It is as if the brain assigns numbers to them according to some mysterious process that we'll call an algorithm, chooses the highest number according to that algorithm, and then acts. This is true of other thought processes as well. Even as I sit here and type this book my mind is in its "write a book" algorithm. Every time I write one word my mind begins working on the next word, discarding those it does not like, keeping those it does like. Ninety-nine times out of 100 it will do this automatically, and my success or failure as a writer is entirely dependent on whether my mind's "write a book" algorithm is a good one or a bad one. In that one time out of 100 when the process is not automatic, when I get stumped trying to express myself, it is not that my algorithm stops working, it is just that a little more processing time is necessary before continuing. Please stand by while new material is sorted through and—yep! There's the right word, and on we go again. Now, anybody can sit down at a typewriter and knock out a series of words. (Who am I kidding? You know damned well I'm working at a word processor.) What separates a good writer from a bad writer is how each individual's "write" algorithm works. Some obviously work better than others. But we all have them, whether or not we choose to use them and regardless of how good they are. And this sort of analogy covers every thought process.

Our ultimate problem as game programmers is the creation of topflight algorithms that will quantify all the data pertinent to the game, giving us credible courses of action to follow in terms of the strategy and tactics of that game. This is the creative process of game programming. We must somehow translate everything we know about a game into a series of rigid computer statements that are completely arbitrary and not the least interested in gaming and end up with a program that not only will play a given game but also will play it well and that will be fun for the human participant in the bargain. To illustrate how this is done I have chosen the game of poker and written a program in which the human player squares off against four different opponents, each of which is played by the computer. Each opponent employs different tactics of play, and each opponent approaches the game with a different strategy. One of those players comes very close to playing perfect strategy, the other three are progressively less good at the game. But more about that later. The point is that within this very long and complicated program are the algorithms that control both the tactics of play and the strategies. The nice thing about poker is that it can be arbitrarily broken down into these two aspects very neatly for the sake of our discussions, because what one does with one's cards is one's tactics, and what one does with one's money is one's strategy. Not all games break down so neatly.

In the world outside of this book the term "algorithm" tends to be used much more freely to describe just about any solution to any computer problem. That would mean that in poker I have an algorithm to shuffle the cards, another to deal, and so forth and so on—which is fine, but I like to see the use of the word confined to the really major problems; and that's how I usually will intend it from now on.

There are no hard-and-fast rules for the creation of algorithms because algorithms tend to be unique solutions to unique problems. What you try to achieve in your game algorithms is the creation of a sophisticated player, an opponent to keep you on your toes. We will do our best in the rest of this book to give you a solid background in the techniques and thinking necessary to achieve this goal.

GOING ALL THE WAY—WHAT MAKES A GOOD COMPUTER STRATEGY GAME

It is all well and good to say that you are going to program a poker game, but just how good a game are you planning on designing? I have seen so-called poker programs where the human player goes head on against the machine—in effect, two-man poker. Who ever plays two-man poker? In the real world poker is a contest among five to seven players; I chose five mostly because that number fits best most conveniently using graphics on the 80-column screen of the Model 4. With a bit of compression you can easily fit these five on the slightly smaller Model III, and using an all-text display you can fit them on the Model I. There's no reason why you can't throw two more players into the program if you are so inclined and you readjust the screen to accommodate them. The point is, if the game isn't realistic right off the bat, it's already got three strikes against it. The primary condition necessary for the creation of a good strategy game is a total commitment on your part. If you really want to sit down and knock off a good poker or blackjack or bridge or cribbage or backgammon or whatever, you've got to steel yourself for a lot of work. It is not an easy job; working in your spare time, you could take from six months to a year to get a good one up and running. The satisfaction when you are finished will be enormous, and by the time you are finished you probably will have learned everything you'll ever need to know about programming. For me, these results have made this long, involving process very rewarding, and I admit quite frankly that as soon as I finish writing this book I plan to get reacquainted with my wife right after I find out what she named the baby.

By now you realize that it is impossible to program a game unless you know that game inside and out. This does not necessarily mean that you as a player are unbeatable at that particular game; even the best of humans tend to err. Since the computer does *not* err, if you've taught it everything you know it will most likely beat you over the long term, a fascinating prospect when you think about it. So pick a game you know and one you enjoy, because you're going to have to live with it for a long, long time.

As you develop your program, it is essential that "completeness" become your credo. As time goes on and you're working like crazy and you still don't seem to be getting anywhere, the temptation to begin cutting corners becomes very strong. One of the first programs I wrote when I was starting out was for seven-card stud. Every time the players got a new card the hands had to be reevaluated, which meant a cumulative effort geometrically progressing in complexity to figure out the value each time of what the players were holding. At the end I finally gave up, and after the last card was dealt I programmed a line that said:

```
I CAN'T SEE THE CARDS IN THIS LIGHT. PLEASE ENTER THE
NUMBER VALUE OF YOUR HAND
```

followed by a choice from zilch to straight flush. How lazy can you get? Ultimately, this was quite a bum game. But after a couple of months I was tired of it and ready to move on. My time programming wasn't wasted—I learned a lot—but when I was finished I didn't have anything to show for it. Be prepared for similar frustrations.

The term "whistles and bells" is used a lot by professional programmers, and it's a concept to keep in mind in your designing. Whistles and bells in a program don't do anything, they simply blow and ring to put a little pizzazz into things. In "Space Derelict!" the occasional beeps of SOUND and the use of reverse printing fall into the whistles-and-bells category. In "Five-Card Draw" our whistles and bells will be the use of graphics in our card display; these graphics are not essential, but they're nice to have. Keep in mind the idea of whistles and bells, and incorporate them into your programs freely. Don't get too carried away with these nonessentials, but a little bit here and there can make for more fun in the playing.

You probably will find after you have finished your program that there are some slow spots in it, where the programming was a little bit loose so you find yourself sitting around waiting for things a little longer than you'd like. Once again, you may be so tired of the program at this point that you simply will accept its pokeyness, but if at all possible you should keep at it and take the necessary steps to get it streamlined. We will be discussing some of these steps as we go on, although I was forced to skimp on a few of them in poker to keep the program readable. But I'll show you the ways speed can be improved, or if not improved at least hidden so no one will realize how slowly the processing is going. It's all right to be sneaky; if you get away with it, great, and if you don't, people will nonetheless admire your cleverness.

Finally, despite everything I've said to the contrary so far, it is important to remember that unless you are a full-time programmer, it is practically impossible to create a program as good as you'd like. So part of your training will be to learn how to create workable algorithms that, even though they do cut an occasional corner, play well regardless. This requires a bit more sneaky creativity, but I've said it before and I'll say it again: If it works, do it.

THE GAME OF POKER

Why choose poker as an example of strategy games for inclusion in this book? A good question, and I'm glad I asked it.

First of all, knowledge of the game of poker is pretty widespread, and I knew that even if you weren't a dyed-in-the-wool poker buff you would know enough about the game to be able to follow the discussion of the program. But the odds are that you *are* something of a poker buff: I have read some vague statistics that indicate that eight out of seven people in America play poker a minimum of six Fridays a month—or something like that. Poker is quite definitely the national card game, so what better subject could we possibly attack?

There is another advantage of poker, and that is that there are enough variations to choke a horse. So after you've absorbed the program in this book you can adapt it to your own favorite poker games; not as malleable as the adventure module, perhaps, but still adjustable.

Finally, I plain wanted to write this program. I enjoy poker and lately have lost the proximity of my regular monthly game. So I've re-created that game exactly, and now it's Friday night whenever I want it, with the chips stacked neatly, the beer chilling in the refrigerator, and all the players eager to get started. That's a big part of what computer gaming is all about—the willing player, whenever you want it.

Besides all of this, there is something within me that says that if you can master all the Basic techniques we'll be using in this poker program, you'll be able to master just about everything that will ever come up in any kind of strategy game. The ultimate idea is to prepare ourselves for any programming challenge we might face; whereas the adventure module is just there, waiting for us to fill in the blanks, for a strategy game (other than a variation on the poker theme) we start with a tabula rasa: you'll have programmed poker, so now try Parcheesi. The important thing here will be to follow my thinking rather than my program.

The game of poker, while simple in some ways, is quite complicated in others. Perhaps that is its attraction as a popular pastime. It's easy to learn and play, but it contains incredible nuances, and it's a lot of fun. Wouldst that all games fell into that category! In terms of game theory, poker adds a new twist to what we were discussing earlier in this chapter—the element of chance.*

In a game like chess, there is no question that the person who plays the better game will emerge the victor. The only factors involved in the play are the abilities of the two players, and the game rests on their strengths and weaknesses. But in poker, the deal of the cards is a big factor. Although it is not inconceivable that you could beat a good hand with a bad hand—that's what bluffing is all about—in normal play over the long term good hands tend to beat bad hands more times than not. The "straighter" a poker game is, the truer this is. In five-card draw, it is about as true as it can get. In Hold 'em, which is played in the World Series of Poker in Las Vegas, this is much less the case. Hold 'em, a version of stud in which the players share three up cards, is very much a game of betting strategy and bluffing. Since so much information is available to the players and since they even share the majority of the cards, all the play concentrates on the hole cards, the ones the other players can't see. There is no draw; one simply plays the cards one has. Very much a betting game.

Straight five-card draw, jacks or better, requires the ability to understand what the game of poker is all about and also the ability to bet a better game than your opponent. Both skills are roughly equal in importance, and one of the joys of poker is that the latter skill changes with the opponents you face, and even changes over time if you always face the same opponents. No two players play the same way; much of one's success at the game is measured by how well one can read the other players. Even if you play with the same opponents year in and year out, each player will continue to grow and change in ability. Again, some of the infinite attractions of the game.

*I can't resist my favorite W. C. Fields line, as he sits down to play poker with a friendly Indian. Indian: "Is this a game of chance?" W.C.: "Not the way I play it."

The first half of your ability to play decent poker requires only that you understand and trust implicitly the idea that every time you flip a coin the odds are 50 percent that it will come up heads and 50 percent that it will come up tails. This is what we all learned in high school as probability. If a coin comes up heads 100 times in a row, the odds are still 50 percent that it will come up heads the 101st time (assuming the coin isn't rigged). This is true; we all know this—and we'd all play better poker if we remembered it all the time. "Luck" has nothing to do with it. Just because we've been doing well all night does *not* mean that our odds of drawing to an inside straight are improved. The odds of filling an inside straight are *always* 4 in 47—four chances of finding a card of the right value in the 52-less-5 that comprise the universe outside the cards we hold. Roughly 1 in 10, pretty shaky odds at best. The good poker player avoids the inside straight like the plague. Let's take each possible poker hand and look at the possibilities.

First there is the zilch hand, no pair, no more than two of any one suit. The best we can do is keep the two highest cards and hope for the best. The odds here are not so bad, actually. Given that there are three each of the two cards we're keeping still outstanding in the deck, we have a 6 in 47 chance of hitting a match. (For this discussion I'm going to forgo the complications that might ensue—in this case, for instance, you *could* draw two of one and one of the other for a full house, or three of one for four of a kind, and so forth, but the possibilities for this are fairly insignificant and are for the most part constant for all the hands. Let's try to keep it simple.) About 1 in 8 is a tad high but not ridiculous. The real problem with this hand is that you are holding nothing and shooting entirely at birds in the bush. How much money should you bet on such chimerical possibilities? None, if you're going into my wallet.

Next you have your three flush. You have a 10 out of 47 chance of getting one match, *compounded* with another 9 out of 46 or the second match. Somewhere around 25 to 1, since you have to multiply the two odds figures (that's another thing they taught in probability class, that when two possibilities are dependent on each other, the probability of getting both possibilities is not the sum of the two possibilities but the product of their multiplication). You've got to be crazy to play a three flush, and a three straight is about the same, so you have to be just as crazy for that one.

A four flush, of course, gives you 9 out of 47 possibilities straight off, which isn't bad, especially compared with the payoff of a very high hand if your ship comes in. A four straight, not inside, is a bit higher, 8 in 47 (filling either end), but still worth the push for the rewards of success.

Hands of a pair or more tend to play themselves, with two exceptions. First, there's the low pair when you also have a four flush or four straight (for instance, the 7 of spades, the 7 of clubs, the 2 of clubs, the queen of clubs, and the 5 of clubs). What do you do in this situation? Well, your chance of improving the pair is higher than the chance of improving the four flush—you could get either another pair or three of a kind with more likelihood than that fourth flush card. How do you play it? How much of a gambler are you? It would really depend on how the betting has been going and what you feel the other players are holding and what you'll need to beat them. Very subtle, very difficult. The second exception to pairs playing themselves is the "kicker"—i.e., holding an ace when you've got a pair or three of a kind. Suffice it to say that your odds are slightly lessened when you hold a kicker, but you may make up for this in the value gained of confusing your opponents as to what you're holding. Again, a fielder's choice, you vs. your opponents.

Despite the logic of what to do and not do vis-à-vis the odds, not all players follow the party line all the time, either because they don't know, or because they are trying to trick their opponents. The first thing we have to consider when writing a poker program is how well the game will play for the person sitting at the keyboard. One of the things we can do in five-person poker is put four opponents up against the player and give each of these four opponents different abilities, which is exactly what I've done. This goes back to the basic concept of making a computer game fun for the player. If all four computer opponents played exactly the same way, it wouldn't take long for our player to figure out what the computer will do in any given situation, since he has four examples of it in front of him all the time. But if each opponent plays differently, not only is our player figuring out how best to play his own hand, but also he now has to unravel four distinct opponent "personalities." The value of the game is multiplied accordingly.

But all of this is concerned with what we've called the tactics of poker, the playing of the cards. What about the strategy, or betting? Well, here we have to take a somewhat different approach, as you'll see when we get down to the brass tacks of the programming. Again, all four players play differently, their betting strategies roughly analogous to their tactical abilities. But here we have to be much more conscious of the human player. Ultimately our betting strategies are static: They are etched into computer stone and, unlike the tactics that are etched into the same stone (a player can never be sure of the unseen cards in the opponent's hand), once a player knows all four computer strategies, he might be able to construct his own strategy to beat the program regardless of tactics. If an opponent always bluffs at a certain time, if an opponent always bets a certain unique amount with a given hand, if an opponent always folds when the player bets a certain amount, the player gets an upper hand that can destroy the fun of the game. So we prepare for this as best we can by making our four strategies as complicated as possible. However, we must face the inevitable, that any program we create is unlikely to stand up against the determined player forever. But we do our best, and that's what it's all about.

8 PLANNING THE PROGRAM

STRUCTURED PROGRAMMING?

There are two words that can strike terror into the hearts of the most valorous computer programmers. These two words can be seen to represent everything from the Holy Grail to the shackles of oppression that must be broken. For true believers, the structured program in Basic is the unattainable; even the best of hackers will coyly comment that the program they've just written is pretty good but not really structured. Academics will insist that structured programming is more important for its platonic value than as a practical aid. Iconoclasts (sometimes but not always including this author) say that if it works, do it—but will think in terms of structured programming when the situation calls for it. And in the business world, in many situations structured programming is a given because the nature of the beasts preclude any alternative. But what do these two words mean, to arouse such varying yet universally strong reactions?

As you know, the digital computer operates entirely in terms of on-off. Electrical impulses flash beneath your keyboard at infinitesimally brief speeds, and the processor performs certain operations on the basis of the content of these impulses. These impulses are either on, an arbitrary value of 1, or off, an arbitrary value of 0. The binary number system, which contains nothing but 0's and 1's, is the mathematical measurement of what the computer is doing. The computer's native language is binary.

Binary numbers look like this: 10110010. For the sake of simplicity (trust me), someone once decided that numbers that look like this—2C, FF, A0—are easier to handle than binaries, so the hexadecimal number system is the one that the computer most regularly speaks to human beings. Fortunately, for us real dummies, the computer can speak decimal numbers as well, but that's two generations away from the native system, which means that occasionally translation problems can arise and lead to difficulties. For instance, 64K does not really translate into 64000 bytes, but we all know what we mean and we're close enough, so nobody really faults us for our third-generation translated approximation in this case. In other, more technical situations, this translation could be a problem (but fortunately one that need not concern us here).

Computers do *not* speak Basic, COBOL, Fortran, Logo, or any other so-called computer language. These languages tend to be cooked up in the basements of name-brand universities or the like, and their purpose is to provide a common ground for communication between the computer and the human. The more humanlike that language is, the "higher" the level of the language. To get really down and dirty, the lowest-level language is binary machine language, which no one really speaks except the computer. But with the use of something called an assembler a human can come reasonably close to binary functions insofar as they are translated into a language of hex numbers. Z-80 assemblers use key words such as POP and LD (which humans can learn to understand) and then create (or assemble) hexadecimal machine-language programs from these key words. Basic, on the other hand, is a high-level language, and in many cases it reads like plain, simple English. You don't have to know anything about programming to understand the line:

```
PRINT "CALL ME ISHMAEL."
```

or even:

```
X = 100: PRINT X/20
```

IF, AND, OR, GOTO, END, CLEAR, etc., are all good human words, at least for English speakers, and in many cases program statements that incorporate these Basic keywords are easily understandable to the uninitiated.

But how does Basic work inside your computer? Well, there's an imaginary little magician inside there known as an Interpreter, which you loaded in when you booted your disk Basic. When you write a Basic program the Interpreter translates each line into a series of hex numbers; for the most part, this is a literal translation, and the result is most definitely *not* a machine-language program: It is a Basic program written in hex numbers, period. When the program is run, the Interpreter takes each line as it comes, turns it into a machine-language operation the computer can understand, the computer performs the operation, and then the Interpreter reads the next line, and so forth. From this you ought to be able to see why a Basic program runs more slowly than a machine-language program; each line first has to be translated, the process continuing one line at a time, whereas the machine-language program simply performs the operations without the Interpreter in the middle.

There is one way of getting around this problem of translation slowness, and that is to use something called a compiler. Compilers are common in large businesses; a business-oriented language such as COBOL, for instance, is actually a compiled language, as compared to Interpreted Basic (although compilers do exist for Basic programs, they are quite expensive and perhaps not all that useful for hobbyists). A compiler is a program that literally rewrites a high-level-language program into a machine-language program (it does some other things as well, but they are beside the point to us—we're simplifying what follows a bit to zero in on our own concerns). After the computer people in your company spend three years futzing around writing a payroll program in high-level COBOL, they compile it. Then the compiled program, now in speedy machine language, is the program actually run on the mainframe computer. Since this program is the one responsible for making sure your paycheck arrives punctually on your desk at its appointed time, you can see why speed can be useful. It might take 24 hours to run this program and get all those checks out every two weeks, and that's the fast version. If the program were not compiled, you would probably be paid your weekly check every three months or so, which you might find inconvenient, to say the least.

Can you imagine that payroll program, that is, the uncompiled COBOL version? You've seen those business printouts of hundreds of pages of code; you know how data processing works in business where one person works on one part of a program, another person on another part, and so on. Can you imagine the complications not only of writing but also of debugging this monster? Hold that thought for a minute.

Structured programming has many purposes. The concept arises in part from the needs of professional business programming. What is the best way to write a program that will lend itself to all the problems writing these monster programs can lead to? Since there are usually about 100 different ways in any language of writing a program to perform the same chore, and we have about 100 different programmers working on the same program, how do we set a plan for the work so that everything each individual programmer is working on fits in with the other programmers' work? We need an overall structure from which the individuals can branch and do their own work however they're doing it, a structured program wherein one big monster is broken down into manageable parts so we never really have to face the whole monster at once.

The same holds true for us when we single-handedly plan a major program. Even though I am one person with one style, this program is going to take me months to complete. Once I'm halfway through it I'll have long forgotten what I was doing at the beginning. If I attempt to create one entire monster all of a piece, that monster will ultimately devour me. But if I structure my program into manageable pieces, I will never have to face the monster all at once, and I will beat that demon and reign victorious for all to see! Or something to that effect.

Further, there is the issue of sheer efficiency for its own sake. Structured programming breaks down the computing process into meaningful segments based on what exactly is going on in those segments. Some high-level languages, such as Pascal, are actually structured programming languages, which Basic is not. So when uninitiates look at a Pascal program, not only can they recognize and understand some of the English commands as they can with Basic, but they can also see to some degree just how the program works: its total structure, its processing flow, is fairly easy to comprehend even without a knowledge of the language. This is definitely not true of Basic, where although the words might make sense, there is no guarantee we'll understand how a program works just by looking at it (even if we've written it ourselves).

But let us backtrack for a minute. With a noncompiled language such as Basic we have the additional problem of that slow process of line-by-line interpretation. To be good programmers not only do we need to understand the structure of a program but also the structure of our language. Since we can't compile our work, what can we do to speed up the processing? Programming structure and language are two different things, but we must think of them simultaneously if we wish to create satisfying Basic programs, especially games. It is no fun to sit in front of the computer twiddling our thumbs while we wait for a particularly convoluted piece of programming to process. So we'll no longer differentiate between the two structures. We'll just lump everything into the concept of structured programming and give the academics another reason to throw this book out the window. In unstructured Basic, structured programming is what we make it.

SOME BASIC PRINCIPLES

For us, the simplest definition of a structured program is one consisting of a series of small subroutines, each subroutine performing one complete part of our overall task. The skeleton of this kind of program might look like this:

```
10 GOSUB 1000: REM INTRO
20 GOSUB 2000: REM INITIALIZATION
30 GOSUB 5000: REM COLLECT USER INPUT
40 GOSUB 10000: REM PERFORM CALCULATIONS
50 GOSUB 20000: REM DISPLAY RESULTS
60 GOTO 10: REM DO IT AGAIN
1000 & ff INTRO and RETURN
2000 & ff INITIALIZATION and RETURN
5000 & ff INPUT and RETURN
10000 & ff CALCULATIONS and RETURN
20000 & ff RESULTS and RETURN
```

Somewhere in the program, in this case lines 10–60, is a series of commands directing processing to the various subroutines, sort of like a traffic cop directing the flow of the program. And while the same data might be used in more than one of the subroutines, each subroutine performs an entire and unique operation. The value of this is twofold. First, when you sit down to program, you have to handle only one subroutine at a time. Second, and perhaps more important, when you debug you can extract each subroutine and concentrate on it rather than having to find your way through the whole program. After you've got one section completely oiled you can move on to the next. Even when you have the whole thing up and running and those last few secretive little bugs finally pop to the fore, this compartmentalization will make it easy for you to recognize exactly where in the program the problem lies so you'll know where to go about looking for it. Another name for this sort of structure is "top-down" design.

The first step in planning a program of this nature is to make a list of all the things you will have to do in the program. Once upon a time even Basic programmers used to draw flow charts with pretty little boxes and circles and diamonds, but I understand that this practice is falling by the wayside, and I'm sure that I've never seen any hobbyist drag out a template and start drawing. A nice legal-size yellow pad is my own favored scratch sheet, and whatever you find convenient is what you should work with. Let's try an outline for tic-tac-toe first. If you're really feeling energetic, don't read what I've written here until you've written out a little outline of your own.

```
GOSUB INTRO—EXPLAIN RULES
GOSUB INITIALIZATION OF VARIABLES
GOSUB DRAW T-T-T GRID ON SCREEN
GOSUB DETERMINE WHO GOES FIRST
GOSUB COMPUTER'S MOVE or
GOSUB PLAYER'S MOVE
GOSUB DRAW MOVE ON GRID
GOSUB FIGURE IF GAME IS OVER
IF GAME OVER THEN GOTO ENDING
IF GAME NOT OVER THEN GOTO NEXT MOVE
GOSUB ENDING
CLEAR GRID AND GO BACK TO DETERMINE WHO GOES FIRST
```

The flow here looks simple enough. First an introduction subroutine followed by initialization. Then the grid is drawn on the screen. Then we determine who goes first; this could either be at the player's option or else simply alternating back and forth between player and computer. With this information in hand, we know whether to take the GOSUB where the computer makes a move or the one where the player makes a move. Whichever it is, next we draw the move on the screen (our player's move subroutine would have covered the possibility of the player making an illegal move). Then we figure if the game is over. If not, we go back to the appropriate subroutine for whoever has the next move. If so, we go to the ending, perform the necessary congratulations or raspberries, and start again.

In the actual program, all this movement would be traffic-controlled just like our skeleton program, with all the real action taking place in the subroutines. We would then proceed to write one subroutine at a time. You could either debug each subroutine as it is written, or you could wait until the end and debug each routine with the whole already laid out in front of you; in either case you would still be concentrating on the individual subroutines. It all looks very easy, doesn't it? In a way it is, and that's the whole point of structured programming.

One thing we're overlooking, however, is a problem I mentioned in the first section of this chapter. You'll remember we talked about the Basic Interpreter and how this quite wonderful little marvel in fact kept slowing us down, taking every line of programming one at a time. And, too, you'll remember from the first part of the book that every time a GOTO or GOSUB is encountered in a program, the processor goes all the way back to the beginning of the program until it finds the correct line to branch to. This, of course, is the fault of the Interpreter. It's one thing to talk about what's really going on in a Basic program, and another thing altogether to look at one. So let's show and tell for a minute.

Let's begin by creating and saving a new program we'll call TESTA.

```
10 CLS
20 FOR X = 1 TO 10
30 PRINT "JIM"
40 NEXT
```

A classic if I ever saw one. Now, let's use an amazingly obscure TRSDOS system command, LIST TESTA (HEX), which is absolutely useless to us as Basic programmers but nonetheless will teach us a thing or two worth knowing. A system list in hex will show us what our Basic program looks like to the Interpreter.

```
0000:00 = FF 13 7F 0A 00 9F 00 24  7F 14 00 82 20 58 20 F1
0000:10 = 20 12 20 CF 20 0F 0A 00  30 7F 1E 00 91 20 22 4A
0000:20 = 49 4D 22 00 36 7F 28 00  83 00 00 00
```

So what do you think of that? If you were to actually type in this LIST (HEX) command you would additionally see some loose translation of the program into Basic on the extreme right, but that information is spotty and beside the point for us. Anyhow, this is our TESTA Basic program the way the computer sees it. Can you believe we actually wrote that ourselves? Well, in fact, we didn't; the Interpreter did the work for us. Let's look at it.

This is a listing of the machine addresses from the start of a Basic program to the end, in this case a total of 43 addresses or, in hex, 2BH. (For the Z-80 hexadecimal numbers are suffixed with an "H"; it is also common to see hex numbers prefaced with dollar signs, e.g., $2B—this is because they are more expensive than decimal numbers.) The first thing we see is the value of FF, which essentially tells the computer that the program begins forthwith. This is followed by the two bytes 13 7F. These two numbers are, in fact, the number of the address in memory where the *next* program line listing will begin. This address is 7F13H (or 7F 13—the Z-80 reads the bytes backward, which is why the Z-80 is the computer and you're the human). Immediately following this information we see the two bytes 0A 00. Again this is backward, and this is actually the first line number listing you would see in the Basic program, in this case the number 10, the decimal equivalent of 000AH. Let's skip then to 7F13H—this address is the eighth one on the first line, the number 24 (keep in mind that hexes go like this: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, etc.). Again we get the number of the next address, 7F24H, followed by the program line number of this listing, 14H (20 in decimal). Let's skip again to 7F24H (which is 24H minus 13H, 11H or 17 decimal bytes away), which gives us the next address as 7F30H. 7F36H delivers us to a 00 value, which means the end of the program. Whenever you do a GOTO or GOSUB, you've got to go through all that; that's how the computer knows where the lines are. Certain information about your Basic program can be and is stored in other places around the computer, but as you can imagine, there is no shortcut for GOTO and GOSUB searches: There is no way that the program line equivalent of every line could be accessed outside the actual program. It would be redundant and would take up half again as much memory as the program itself. So we have to get around this limitation, and the best way to do it is never to use GOTO and never to use GOSUB. Impossible, you say? All right, we'll compromise. You can use GOSUB, but we'll put the more frequently used GOSUBs at the beginning of a program so we won't have to go through all this rigamarole at every access. Of course, you still can't use GOTO, but more about that in a second.

Let's go back and look at our tic-tac-toe. Some of these subroutines would come up only once in a program, but others, such as the figuring of moves and figuring the state of the game, would keep coming up again and again. So what we would do is put these frequently accessed subroutines at the beginning of the program, where the Interpreter can get to them as quickly as possible. Makes sense, doesn't it? We'll talk more about this when we see how it was done in the poker program. The logic of it should be clear, however, at this point.

But what about those GOTOs? Hell, "Space Derelict!" was positively a breeding ground for them, and it ran fine. Very true, but the number of calculations in that program from the moment a player enters his input until the results are seen on the screen is very small. Computers are by nature fast, remember, so sometimes we really can't see the delays we're creating. But in a game like poker, or most likely any strategy game you create, you will be setting your computer on some fairly long and involved calculations, and there will be a noticeable pause every now and then while the computer does what you are asking it to do. The point of minimizing GOTOs is to keep these pauses down as much as possible. Also, the use of GOTO can lead to what is so ingloriously referred to as spaghetti logic, wherein a program resembles a plate of the same, and it becomes very hard for anyone, including the programmer, to figure out what he is doing or where he is going. "Space Derelict!" does have its share of spaghetti logic, but in that simple sort of program, where there isn't that much spaghetti in the first place, it doesn't matter very much. But in a complicated program, you want your logic as clear as possible. The normal temptation is to program yourself into a corner and then program yourself out of that corner, whereas what you should do when you get into that corner is simply throw away what you've just done, rethink it, and start again. So we can't, of course, dispense with GOTO altogether, but we can use it carefully.

One thing always to be on the lookout for is the mob and the leper at the turn in the road. There is a sign at this turn: "Lepers turn right!" We want to make sure we don't have the mob going right, as this will only slow them down while they read the sign and try to figure out which direction to go in. Translated into programming, in the flow of what you are doing you will always be breaking down your information into patterns and variables, and after the breakdown is finished some of these variables will go in one direction while others will go in another direction. You want to redirect the lepers, the minority, while the mob continues along on the most direct route. For example:

```
10 INPUT "PICK A NUMBER FROM 1 TO 10";X
20 IF X < 10 THEN 100
30 PRINT "YOU PICKED THE NUMBER 10"
40 etc.
```

This is an example of meatball thinking, which is this book's answer to spaghetti logic. What we're doing here is directing the mob while the leper goes through in a straight line, whereas the odds of getting the leper are only 1 in 10. Doesn't the following make much more sense?

```
10 INPUT "PICK A NUMBER FROM 1 TO 10";X
20 IF X = 10 THEN 100
30 etc.
```

In this case our sign at the turn is directed at the leper, while the mob moves along in a straight line. This analogy may be a tad strained, but I like it, and however you choose to visualize it, it is important to remember.

There's one other thing to keep in mind, and I want to go back to the TESTA program above. Let's see what would happen if we tightened it all into one line thusly:

```
10 CLS: FOR X = 1 TO 10: PRINT "JIM":NEXT
```

Let's list this one in hex and see what we get.

```
0000:00 = FF 27  7F 0A  00 9F 3A 82   20 58  F1  12 20 CF20 0F
0000:10 = 0A 3A  91  22  4A 49 4D 22   3A 83 00   00 00
```

Even counting the initial FF and final 00s, we've cut the program by 15 bytes, roughly one third. In the process of doing nothing other than combining likely multiple lines into one long line with statements separated by colons (and ELSEs work much the same way), we have killed two birds with one stone: We have taken up less space in the program, and we have put in fewer addresses for the Interpreter to sort through when the necessary GOSUBs and GOTOs are encountered. The message is obvious: If you can put multiple statements on a line, do so. The value is enormous, and we'll be doing it as much as possible in our poker game.

THE POKER FLOW

Now let's get down to the business of the poker program. The first step was a work-up of just how the program would flow; that work-up was created in about three minutes on a piece of yellow legal-size pad, and looking back on it now that the program is finished, it is surprising how well it held up. But that was the point of it in the first place, so maybe it's not so surprising after all.

When working out a program flow, simply picture in your mind all the progressive steps of the game you wish to program. Sometimes you'll end up with a very vague outline that won't take into consideration some of the unforeseeable thorny problems; at other times you'll find yourself anticipating just about every complication. Either way, you'll face the problems in actual programming sooner or later, so it doesn't matter much how detailed your preliminary outline is at this point. Do it however you feel the most comfortable.

Five-card draw is a relatively simple game in terms of moves. The players are dealt their cards, they bet, they draw some new cards, they bet again, the winner takes the pot. Seven-card stud would have been a much more complicated program because the players get dealt three cards, bet, get another card, bet, and so on up to seven. After each card is dealt, the program has to reanalyze the hands, and the programmer has to know all the moves and variations from three cards up to seven—a lot of work and a lot of programming. Also, in our five-card draw we are going to insist on jacks or better to open; this means that we need not concern ourselves with creating betting algorithms in the face of sheer guesswork. When you play draw as ‘‘anything opens’’ you end up with a fairly wide-open game. To program this you have to create an algorithm that not only would evaluate a hand but also would decide whether to open on that hand. In real life the factors guiding that choice are complex. If you have a good hand but are sitting too close to the dealer's left, you might pass in hopes of sandbagging—raising someone else's opener. This sort of decision rests almost entirely on player psychology, which is why it becomes very difficult to computerize it. But we're avoiding it entirely with our jacks-or-better game.

This is how my flow looked originally:

```
INTRO
SEED RANDOM NUMBERS
INITIALIZATION OF VARIABLES, HOUSEKEEPING
FIGURE WHO DEALS & WHO'S STILL IN GAME
ANTE
SHUFFLE
DEAL
FIGURE HANDS
ASSIGN VALUES TO HANDS
FIGURE IF THERE'S AN OPENER
IF OPENER THEN SKIP TO FIRST ROUND OF BETTING
WHEN NO OPENER HAVE SPECIAL NO-OPENER ROUTINE AND GO BACK TO FIGURE WHO DEALS
FIGURE BETTING PATTERNS
FIRST ROUND OF BETTING
GET DRAW CARDS
FIGURE HANDS
ASSIGN VALUES TO HANDS
FIGURE NEW BETTING PATTERNS
SECOND ROUND OF BETTING
DECLARATION OF HANDS
FIGURE WHO WON
CLEAR VARIABLES AND DO IT AGAIN
```

For the most part, this is self-explanatory. The first step should be familiar by now, as should be the third; we'll talk about the second one in the next chapter. You can count on these three in any strategy game with a random factor (in this case our deck of cards), and two out of three any other time. Some of the steps that follow, while still somewhere in the final program, turned out to be unnecessary; the fourth step, figuring out who's dealing, is one of these, and when I finally attacked it this became a very simple matter and needed only a line or two and not a whole routine. The round of betting where there is no opener ultimately became absorbed into the standard first round of betting when I started streamlining. The step ‘‘figure hands’’ means coming up with a variable for each possible hand that would stand us in good stead when we came to the draw. For example, if a hand was ‘‘P23’’ that would mean we had a pair, the second and third cards of the five in the player's hand. Assigning values to a hand meant simply giving the hand a number in the hierarchy of hands from zilch to straight flush; our ''P23'' eventually would have a value of 5 (we'll go into all this in detail later, so don't worry about it now).

This list on my yellow pad was all I had to go with as I began programming. The first thing I did was translate this list into a series of GOSUBs, as in

10 GOSUB INTRO
20 GOSUB SEED

etc. Then I began writing each subroutine, filling in the blanks, trimming here, cutting there, streamlining, rethinking. This is how the traffic controller in ''Five-Card Draw'' finally ended up (and here's where to begin typing if you want to save the program—the *FIVE-CARD DRAW* legend is what to look for):

Five-Card Draw

```
10     GOTO 10000
10000  GOSUB 60000
     : REM   INTRO AND SEED
10010  GOSUB 50000
     : REM  HOUSEKEEPING
10020  GOSUB 6000
     : REM   SHUFFLE AND ANTE
10030  GOSUB 18000
     : REM   DEAL
10040  GOSUB 90
     : REM  FIGURE HANDS
10050  GOSUB 8000
     : REM  FIRST ROUND OF BETTING
10055  IF DROPFLAG = 1 THEN DROPFLAG = 0
     : GOTO 10020
10056  IF WINFLAG = 1 THEN WINFLAG = 0
     : GOTO 10140
10060  GOSUB 19000
     : REM   CLEAR SCREEN
10070  GOSUB 2000
     : REM  GET DRAW CARDS AND FIGURE HANDS

10080  GOSUB 5000
     : REM  SET NEW PATTERNS OF BETTING
10090  GOSUB 19000
10100  GOSUB 9000
     : REM  SECOND ROUND OF BETTING
10105  IF WINFLAG = 1 THEN WINFLAG = 0
     : GOTO 10140
10110  GOSUB 19000
10120  GOSUB 27000
     : REM  DECLARATION
10130  GOSUB 28000
     : REM  FIGURE WINNER
10140  GOSUB 29000
     : REM  CLEAR THE DECKS AND DO IT AGAIN

10150  GOTO 10020
```

Right off the bat you can see that this traffic-cop routine does not reside at the beginning of the program. The first line of the program is line 10, sending us way up to 10000. What we've done here is put our most active programming down at the early lines; 10000 is roughly in the middle of the program. This is a normal structured programming procedure in Basic, the actual placements depending on exactly what sort of calculations you are asking the computer to perform. Needless to say, the values of the numbers of the program lines are meaningless to the Interpreter; it takes no longer to read the bytes containing big numbers than it does little numbers. That is, if your program goes 10, 20, 80, 9090, 10000, 10010, and so forth, it takes no longer to go from 80 to 9090 than it does from 10 to 20; each decimal number is still represented by two backward bytes of hex, and that is all that matters. The usual practice for convenience's sake is to begin all new subroutines at thousand or hundred intervals, depending on the size of the program.

As you can see, the introduction and seeding are now combined at 60000, followed by housekeeping at 50000. Since these two routines are encountered only once, at the beginning of the games, they are way up in the listing, saving the prime real estate for more frequently accessed subroutines. The two once-separate acts of shuffling and anteing are also now combined, a little better situated at 6000. The dealing of the cards, which is actually the formatting of the screen into five separate hands, happens at 18000; speed is no factor here, and if anything, this routine as written could optionally use some pauses to slow down the somewhat dizzying pace at which the graphics are drawn to the monitor.

The first big gun, way down low at line 90, is the routine for figuring hands, which is quite slow because it forces us to use numerous GOTOs, and to put it simply, the bad hands take longer to process than the good ones. This routine goes step by step, eliminating all the possibilities; obviously, the worse a hand is, the more possibilities that must be eliminated. So this one gets the prime real estate in the program.

The first round of betting (whether or not there's an opener) is next, at 8000. The two flags handle no opener and one player still in, respectively. The clear-screen routine actually does not clear the whole screen, just parts of it. We'll be doing some tricky graphics screen formatting in this program, which, if you can master it, will make you a master formatter, a skill not to be valued lightly.

Getting the draw cards and refiguring the hands become one subroutine, again quite low at 2000. This one is a bit swifter than the first hand figuring, because now at least some possibilities have been eliminated. But again, still prime real estate. Hard on the heels of this GOSUB is the routine for setting the new patterns of betting (the old patterns will have been set in the figure-hands routine at 90—again, your streamlining at work). 19000 clears (part of) the screen again, followed by a second round of betting, another clear, the declaration, the winner, and the clearing of the decks to do it again.

We'll talk more about the rationales for locating the routines when we discuss the routines themselves. For now, the last thing to point out is that apparently (although only theoretically) this all could have been done in one line, thus:

```
10000 GOSUB 60000: GOSUB 50000: GOSUB 6000: GOSUB 18000: GOSUB 90: GOSUB 8000: IF DROPFLAG=1 THEN 10020 ELSE IF WINFLAG=1 THEN 10140 ELSE GOSUB 19000: GOSUB 2000: GOSUB 5000: GOSUB 19000: GOSUB 9000: IF WINFLAG=1 THEN 10140 ELSE GOSUB 19000: GOSUB 27000: GOSUB 28000: GOSUB 29000: GOTO 10020
```

Ignoring the fact that there are some exceptions to this flow (again, to be discussed later), why don't we do it something like this? We could if we wanted to. If we were really that hard up for space in the program, this sort of thing would not have been impossible. But it would have been extremely confusing. A traffic cop in real life stands in the middle of the street where you can see him clearly so you can follow his directions exactly, and that is what we're trying to do here. A tight program does not mean incomprehensibility (or at least it shouldn't, although sometimes it can, which is a problem we must strive to avoid). Clarity is just as important as flow speed, so don't get carried away with this stuff. Make it easy to understand, even if you're the only one who's ever going to have to understand it. Confusing programs are what give programming a bad name.

9
SETTING IT UP

THE INTRODUCTION

The first subroutine in our poker game is the first subroutine in any game, the introduction. The length and complexity of your introduction are entirely dependent on the nature of your game. If, for instance, the game is one of your own invention (something we'll discuss toward the end of this book), you will obviously need very detailed instructions. However, if you're programming a game where you can expect the player to be familiar with the rules, you needn't get carried away. If you tackle bridge, for instance, you could fill up your entire RAM trying to explain how the game is played. The best thing to keep in mind is that if somebody doesn't know how to play bridge, or poker, or gin rummy, or whatever, the odds are he'll never run your program in the first place, whereas anybody who does run that program will already have an interest in the game. So what you need in this sort of introduction is simply to present the pertinent information about your computer version of the game to the knowledgeable player. Let's look at how this is handled in our poker program.

Five-Card Draw

```
60000 CLS
    : PRINT TAB( 35)"DRAW POKER"
60010 PRINT
    : PRINT
    : PRINT
    : PRINT
60020 PRINT TAB( 25)"  Copyright (c) 1984
       by Jim Menick"
```

```
60025 PRINT CHR$ (15) + CHR$ (21)
60030 FOR PAUSE = 1 TO 5000
    : NEXT
60040 CLS
60050 PRINT "These are the house rules:"
    : PRINT
60060 PRINT "    Each player begins with $1
       00"
    : PRINT
60070 PRINT "    1$ ante"
    : PRINT
60080 PRINT "    Jacks or better to open"
    : PRINT
60090 PRINT "    Three raise limit"
    : PRINT
60100 PRINT "    5$ betting limit"
60110 PRINT
    : PRINT
    : PRINT
    : PRINT "Hit space bar to continue"
60120 RANDOM
60130 X$ = INKEY$
60140 IF X$ = " " THEN RETURN ELSE60130
```

Line 60000 gets us to the top of the screen, followed by a printing of the title page of the game in 60010–60110.The TAB command, in that TAB(X), places the cursor in the Xth column from the last printed character. When we're starting at scratch, as we are now, TAB(X) puts us in the Xth column from the absolute left.

Line 60020 is where you get to exercise your designer's ego. Keep in mind that just because you say a program is copyrighted doesn't make it copyrighted. To make it official you must register your program for a nominal fee with the copyright office in Washington, D.C. Which brings us to a few words about getting your programs published.

I read somewhere recently that writing the Great American Program has lately replaced the quest of writing the Great American Novel. Everybody's writing the former nowadays, and everybody wants to get them published. But how do you protect yourself from unscrupulous publishers who will steal your idea and never give you credit for it? Well, one thing is not to send your program to unscrupulous publishers in the first place (even if it's copyrighted, nothing prevents the unscrupulous publisher from stealing it anyhow). If you really have a program you think is salable, the first thing to do is contact the major software publishers and find out if they even consider submissions from outsiders. If so, just send them the program and don't worry about it. These publishers have better things to do than steal your ideas. First, if your idea is no good, they won't want to steal it, and second, if it is good, it's easier for them to publish it than to steal it. They're not in the business of ripping off newcomers. These are honest businesspeople looking to make money, and in many cases they also carry a bit of the pioneer spirit with them, a spirit they are eager to share. Of course, you should be a bit wary of a software publisher you've never heard of before who advertises in ''Bum Biters Monthly,'' but other than that you can feel reasonably confident that well-known publishers are operating on the up and up. If a publisher decides to take on your program, then it's time to worry about copyright.

Line 60025 takes us into territory half familiar and half completely incognita. The first part, printing CHR$(15), is something we did in ''Space Derelict!'' This is turning off the cursor. With our complicated graphics display in the rest of ''Five-Card-Draw'' the last thing we want to see is a blinking dash (or a dashing blink) flashing all over the place. Keep in mind that turning off the cursor does not literally do away with the functions of the cursor; it just makes it so we can't see it. Whether we see it or not it is still there, doing whatever it normally does, which in graphic displays usually translates as messing us up. The second part of this line is the printing of CHR$(21), which, if you look it up in your ASCII Character Set in the manuals, ''swaps space compression/special characters.'' One of the nicest aspects of the more recent vintage TRS-80 machines is their very large expanded use of the ASCII codes. On an Apple II, for instance, you have a few less than 128 characters, and that's it. But on the TRS-80 you've got graphics characters, special characters, fat characters, and even what I call ''special'' special characters (the foreign letters from ASCII 1 to 31, if you use them). Anyhow, CHR$(21) in the present case turns on for us the special characters from 192 to 255, which include most importantly the symbols for the four card suits, CHR$(192) to CHR$(195) being spades, hearts, diamonds, and clubs, respectively. We're also going to use the combined characters CHR$(244)+CHR$(245)+CHR$(246), which together print a nice pointing finger on the screen. We will not, nor will we ever, use the smiley face at CHR$(196). In the course of our game we'll also use the abstract graphic characters from 128 to 191, but we'll talk about them in detail when we get to the chapter devoted entirely to graphics.

But back to introductions. As you can see, the ''Five-Card Draw'' one is brief and to the point. Knowledge of the game of poker is assumed, so the only things you get here are the special points pertinent to the computer version. First, we tell the player that everyone has $100. It is a good idea for you as designer to set the bankroll limitations rather than allowing the player to enter his own bankroll. This puts some preplanned boundary to the game. If the player has an infinite supply of money, or close to it, he loses some of the fun of trying to manage the funds at hand. In this game, $100 goes quite a long way. If you want to make the game a little more cutthroat, change this amount down to $50.

In this program the anteing is handled automatically, which is why I've explained in the introduction how much it is. It could just as easily be shown later on the screen in a statement such as:
PLEASE ANTE $1
A fielder's choice, so to speak.

Jacks or better is the name of this game, according to line 60080. As I mentioned earlier, it was a lot easier to program draw poker this way than with anything opens. And, as a player, I prefer the traditional jacks or better. The only problem we get in the program as a result is that every now and then we get hands where nobody opens, which means sitting at the computer for a while with nothing much happening while the cards are reshuffled and redealt. Unfortunate, but an occupational hazard, just as frustrating in the real game as it is here.

The three-raise limit is not unusual in the friendly Friday-night poker game, but it serves a special purpose here. If we had put no raise limit on the game, our subroutine that handles the betting would have been all that much more complicated: How would the computer figure out when to stop raising? With the three-raise limit, the computer just stops when we hit the third raise. And finally, the introduction specifies the $5 betting limit, which serves two functions. Again, having such a limit helps us in programming the computers betting moves. Without it, how would the computer know when to stop? In effect, this limitation helps give the computer "players" their version of the poker face. The second aspect of the $5 limit is its relationship to the total bankroll of $100: Even the greediest player will hang around for a while at these prices. We don't want the player to go bust too soon, and too high a betting limit might help him to do just that.

Which brings us to random numbers. The first thing we do is print to the screen the message in line 60110, telling the player what is expected of him. Then, in line 60120, we pull one of those neat little hat tricks that Basic is capable of. You see, every time you turn on your machine it is capable of giving you random numbers, but it uses a stack of the same random numbers every time. What you have to do is circumvent this random regularity by using the command RANDOM, as we do here. Doing this "seeds" the random number so that we are not taking the top one off the stack but instead a truly random one somewhere down in the middle. The actual pulling of a random number, now that the stack is ready, is done using the RND(X) command. RND(X) gives us a random number from 1 to X; needless to say, we will be using RND(52) in our program.

Our introduction subroutine ends with the now familiar INKEY$ function beginning in 60130. As soon as we get a space, " ", we RETURN back to the traffic direction routine at line 1000.

ARRAYS MADE COMPLICATED

In our adventure game we had very few variables, and it was never very difficult keeping track of what was what and what was meant by it. Unfortunately, that will not be the case in the poker program, as it probably will not be the case in most strategy games you design. The problem is that you have to keep track of an awful lot of information, each piece of which is independent of the other pieces, so you're forced to bring into play a lot of different variables. But one lesson we have already learned from "Space Derelict!" is to use arrays when the situation is ripe for them. If we have variables thematically related to other variables, bringing arrays into play usually makes sense. In "Space Derelict!" we used D(X) for doors, CO(X) for movable objects, and so forth. For the sake of simplicity, you should try to follow through on this practice.

Another worthy practice is always to use the same counter variables—i.e., always use N as in FOR N = 1 TO 10 rather than using N this time, X that time, and Z the next time. For any counter variable like this where once you use it, it doesn't matter what its value is, you might as well use the same one over and over. The smaller the number of variables, the more memory available to your program.

Not counting arrays, there are about 50 variables in the poker program. Some of these variables come up often during the execution, while others have very specific uses in some of the darker corners. As a rule I've tried to make the variables somehow meaningful to me so I could remember what they were all about, so there's one variable called DUBCHK$(X), another called WINNER(X), and so forth, where obviously I've gone on a little longer than necessary, but the point in these cases was that I could understand what I was doing, a much more important concern than memory space. If you can keep your variables down to one or two letters, so much the better, but it really doesn't take up much more space to throw in whole words like this, and as a program grows in complexity you might find yourself doing it just as I did.

As with an adventure, our variable initialization/housekeeping takes place early in our strategy programs, but since this is a one-time access we can put it way up in the stratosphere of line numbers. As you will see, all of my 50 variables are not accounted for in my housekeeping, even though, as you'll find out later, some of them are arrays. As we mentioned earlier, if an array has 11 or fewer elements (e.g., D(10) or less—don't forget the 0 value), the TRS-80 will automatically set aside this space for us the first time the undimensioned array variable is encountered in the program. As for the nonarray variables, there seemed to be no point in noting these here if no initial value was necessary for them; they'll come up soon enough in the ordinary course of events. Again, I'm trying to save myself programming time—why throw in a bunch of REMs when I don't really have to. This time-saving may be all right insofar as the straight variables are concerned, but there is a trade-off in not dimensioning the 11-or-less arrays. The thing is, even if I don't use all of my default allowance of 11, the computer has nonetheless set aside that much space for them. And variables (especially of the string persuasion) take up space in memory, whether we use them or not; the fewer used, the more space you have for more important things. And don't kid yourself into thinking 64K is more than enough for everything you want to do, because it isn't—not for adventures, and not for strategy games. Sloppiness on a big program can easily cause you to run out of memory, which is why we'll be talking a lot in this part of the book about condensation and space-saving, as well as time-saving in execution. But in this particular instance the actual number of variables set aside but not used is negligible, so we don't have to worry about it.

Rather than bombard you with all my variables right off the top, we'll discuss most of them when we encounter them in the programming. So now let's look at the housekeeping for "Five-Card Draw":

Five-Card Draw

```
50000 REM  SET DIMS AND VARIABLE NAMES
50010 DIM R(5),P(5),P$(5)
50020 FOR X = 1 TO 5
    : R(X) = 100
    : NEXT
```

```
50030 REM   P(X) = PLAYER NUMBER - 0/IN, 1
         /OUT, 10/OUT FROM THIS HAND ONLY
50040 D = 1
50060 P$(1) = "MARVIN"
50070 P$(2) = "GERRY"
50080 P$(3) = "WALTER"
50090 P$(4) = "BETSY"
50100 P$(5) = "YOU"
50110 DIM C(40),C$(40),S(40),S$(40),CD$(40
         )
50115 DIM CH(52)
50120 FINGER$ = CHR$ (244) + CHR$ (245) +
         CHR$ (246)
50130 DIM V(5,5),V$(5,5),ST(5,5),ST$(5,5)
50140 TPCARD$ = CHR$ (151) + CHR$ (131) +
         CHR$ (131) + CHR$ (171)
50145 MIDCARD$ = CHR$ (149) + CHR$ (128)
         + CHR$ (128) + CHR$ (170)
50150 BOTCARD$ = CHR$ (181) + CHR$ (176)
         + CHR$ (176) + CHR$ (186)
50155 GOBACK$ = CHR$ (26) + STRING$(4,24)
50160 ART$ = TPCARD$ + GOBACK$ + MIDCARD$
         + GOBACK$ + BOTCARD$
50161 ERAS$ = CHR$ (128) + CHR$ (128) +
         CHR$ (128) + CHR$ (128)
50162 ERAS$ = ERAS$ + GOBACK$ + ERAS$ + GO
         BACK$ + ERAS$
50170 DIM B$(12)
50190 B$(1) = "I'll open for $"
50200 B$(2) = "I'll see that"
50210 B$(3) = "I'll raise you $"
50220 B$(4) = "The bet is $"
50230 B$(5) = "The pot is $"
50240 B$(6) = "You have $"
50250 B$(7) = ": Opener"
50260 B$(8) = "I can't open"
50270 B$(9) = "I'm out!"
50280 B$(10) = "I'll take "
50290 B$(11) = "I've got zilch!"
50300 B$(12) = "                    "
50320 RETURN
```

Line 50010 introduces our first three variables in the small arrays R(5), P(5), and P$(5). They're accounted for in the following lines. 50020 fills in R(X). In this case each of the five values in the array represent one of our five players, and R(X) refers to the amount of money each player has (the derivation of R, in this case, from bankRoll—I always use R as my money variable)—$100 each across the board.

Line 50030 is a REM I kept because it's extremely important. The variable P(X) refers to each of the five players in the game from P(1) to P(5). The value of P(X) tells us whether that player is in the game (a value of 0), out of the game entirely (a value of 1), or merely dropped out from this hand only (a value of 10). This is how we can keep track of whether to deal a given player a hand, whether he should bet, etc. You'll find:
IF P(X) < > 0

or variations thereof sprinkled liberally through the rest of the subroutines as a check to make sure our particular player is still with us.

Line 50040 is yet another variable, in this case D as in Dealer. We're initializing this variable with the value of 1 to indicate that player number 1 (or P(D)—P(1)) will deal the first hand. Altogether we will have three variables keeping track of who deals, who bets, and so forth, but D is the granddaddy of them all on which all the rest are based. We'll get to the others when we need them.

In 50060–50100 you get to meet the opposition, as well as yourself. As you've probably surmised, there is a one-to-one correspondence between P(X) and P$(X). When we're talking about P(1) we're talking about Marvin; when we're talking about P(2) it's Gerry, and so forth. With the excepton of YOU, these players are all handled by the computer, and as promised, each is handled differently. In fact, all four of them are based on my real-life poker buddies, and to some degree the way the computer plays their hands is similar to the way they play in real life. They were my inspiration in coming up with the algorithms for the computer strategies.

In line 50110 we've got our next set of arrays, each of which has been dimensioned at 40. These five variables are the ones we're going to use to represent the cards at play in the game. The reason there are 40 rather than 52 is that in five-card draw with five players you need a maximum of only 40 cards (25 on the deal, plus up to three for each player for the draw). Here we're saving 60 (5 variables × 12 values) variable space allotments by making sure we're not asking for more space than we need. In many card games you will similarly not use an entire deck at once, and this figuring ahead of time can save you memory space (and shuffling time) later on.

C(X) refers to the value of the card, which will range anywhere from 1 to 14, low ace to high ace. C$(X) is the corresponding name for that value, from the numbers 2, 3, etc., up to A. S(X) refers to the suit; 1 equals clubs; 2, diamonds; 3, hearts; and 4, spades. S$(X) is the equivalent CHR$ character symbol of the suit name. CD$(X) will be the actual drawing of the graphics card on the screen, based on the concatentation of C$(X) + S$(X); if C(4) = 11 and S(4) = 3, then C$(4) = "J" and S$(4) = "♡" and CD$(4) = "J ♡" housed in neat cardlike rectangle. All very elementary, and now you can see how we establish both values and names for each of the cards that will be dealt in the hand.

Line 50115 dimensions a counter variable we'll be using when we shuffle the cards. CH(X) as in CHeck will track for us the consecutive values of the cards as we pick them to make sure each one is not one we've already picked. Later CH(X) will do additional duty, CHecking for us the individual value of each player's hand. We've already mentioned the special characters comprising the pointing finger, and line 50120 is where we initialize them into the text, into the unimaginative variable FINGER$.

Next we get to the hard part, line 50130. We talked all about arrays in Part One of this book but mentioned in passing that there was more to come of a more difficult nature later on. Well, here it is, the double-decker dimension, and in fact it's not all that complicated to understand, but it can be complicated to manipulate. Let's talk about theory first.

You can have virtually unlimited dimensions to an array, provided you don't take up all your memory space—or at least that's what they say in the manuals. But I'll bet no one has ever used more than a handful of dimensions in an array in the history of computing. It's just too complicated to keep track of. But smaller dimensions of two or three elements are not at all uncommon, and their use can be very valuable in keeping track of certain kinds of data.

The best example I can think of to show how the double-decker dimension works is the one used in "Five-Card Draw." In this game we have five players, and each player has five cards. So imagine that player 1 gets his card 1, player 2 then gets his card 1, and so forth up to player 5 getting his card 1. Then player 1 gets his card 2, and so forth. Five players, each with a different card 1, card 2, card 3, card 4 and card 5. Our double-decker array holds that information for us one piece at a time: V(1,1) is the value (same meaning as C(X) above) of player 1's card 1. V(1,2) is his second card, and so on. V(2,1) is player 2's card 1, V(5,4) is player 5's card 4, and so on. V$ is the equivalent of C$, ST the equivalent of S, and ST$ the equivalent of S$.

But what is the reason for all this? In "Five-Card Draw," the double-deckers allow us to use the same subroutines over and over again regardless of which player we happen to be talking about. In these routines we're interested in the cards a player has from card 1 to card 5, and we'll be evaluating these cards and creating new variables that tell us what a player is holding *in which cards*. For instance, not only will we want to know that a player has been dealt a pair, but also we'll want to know which two (e.g., first and fifth) of the five cards the pair happens to be. Use of the double-deckers allows us to do this.

If all of this sounds hopelessly vague and confusing now, hang on a bit until you see it in action. As long as you can understand that V(1,3) means player 1, card 3, at least you understand the concept underlying multiple dimensions. Their use will become clear later on.

We'll just note in passing that lines 50140–50162 are part of our graphics display, a subject we'll cover in detail in Chapter 10. The rest of these lines should be self-explanatory, and you'll see them in use soon enough.

SHUFFLING THE CARDS—MAKE-BELIEVE SPEED

If you pay attention to what's happening in the microprocessor field, you are probably aware of manufacturers' never-ending quest for more speed. On the most dramatic front, advances in processing speed are gained simply by changing the nature of the processor from an 8-bit to a 16-bit, a 32-bit, and so forth. Obviously, the more numbers the computer can bite off in one piece, the more work that can be done faster. And in our discussion of structured programming we mentioned how more careful programming, which takes into account how Basic works in our machines, can make our programs run faster. But although most of the time this concern with processing speed seems superfluous, given the already incredible speed with which any computer operates, sometimes it is a question of vital importance. Sometimes you are going to create an algorithm to perform a certain chore, and although that algorithm is a perfectly good one, it will be too slow to be satisfactory. Let's face it: Given the usual swiftness of data-shuffling that faces us every time we sit down at the computer, it can be very surprising and even frustrating when we're just sitting there waiting for the computer to finish what it's doing. Getting around this problem is what this section of this book is all about.

The first thing to do if you design one of these albatross routines into one of your programs is to make sure it's written as "structured" as you can get it. Are there too many GOTOs in it? Is it at the end of the program when it should be at the beginning? Can it be trimmed down by merging lines, deleting REMs, etc.? After checking all these possible tardiness-makers, if you've still got a slow-moving albatross, perhaps you should try to rethink your algorithm. Would there have been a better way to accomplish the chore that's been slowing you down? Completely throw out that part of the program and see if you can rethink it. Perhaps a speedier solution is just waiting for you to clear your head of your first attempt. Any creative process has its temporary dead ends, and maybe your albatross happens to be one of them. A good library of programs might be helpful to you as well: In a book or a magazine maybe you can find a similar problem, the solution to which you can adapt to your own needs. All these things are worth trying, but there is a last resort when all else fails and you're stuck with your albatross as is.

Despite the fact that I have programmed a number of games incorporating the shuffling of cards, it was only after this book was being typset that I designed a formula for this that I find truly satisfactory. The one or two Basic solutions to this problem that I'd tried to glean from other available programs didn't seem to be any improvement over my own albatross. The problem in shuffling is a simple one: It's easy to generate 52 random cards, but it's not so easy to make sure you're not generating the same ones more than once. Fifty-two *different* cards are needed, and therein lies the rub. Every time the computer generates a new card it has to go back and make sure that card hasn't already been generated; if it has, then the computer has to try again. And as each card is dealt, the harder this becomes, because the computer has to go back over more and more previously dealt cards each time to make sure the new one isn't a duplication. This is time-consuming, even for a computer working at breakneck speeds. And it's time when our player sits in front of the computer and nothing seems to be happening. Not at all a desirable situation.

My penultimate shuffling routine took approximately 15 seconds. By re-thinking it, I managed to get it down to about 5 seconds, a substantial savings. But 5 seconds is still 5 seconds, an interval that seems ponderously long for the player with nothing better to do than stare at the blank screen of the computer.

But there is a way around this problem. After we've got our albatross as fast as we can make it, we simply bury it. We integrate it into the program so that even though it takes the same amount of time to process, the player isn't conscious of that time elapsing. While the computer is tirelessly toiling away at this one chore, we mix in another, superficial activity on top of it so that the player still has something to keep him busy and is never the wiser about the albatross that otherwise would have been staring him in the face.

```
6000  REM   SHUFFLE
6010  CLS
    : N = - 1
6020  PRINT CHR$ (16) + "SHUFFLING" +
        CHR$ (17)
6060  FOR X = 1 TO 40
6070  C(X) = RND (52)
6080  IF CH(C(X)) = 1 THEN 6070
6090  CH(C(X)) = 1
6130  S(X) = INT ((C(X) - 1) / 13) + 1
6140  S$(X) = CHR$ (196 - S(X))
6180  IF C(X) > 13 THEN C(X) = C(X) - 13
    : GOTO 6180
6190  IF C(X) < 10 THEN C$(X) = STR$ (C(X)
        )
6195  IF C(X) = 10 THEN C$(X) = "T"
6200  IF C(X) = 11 THEN C$(X) = "J"
6210  IF C(X) = 12 THEN C$(X) = "Q"
6220  IF C(X) = 13 THEN C$(X) = "K"
6230  IF C(X) = 1 THEN C$(X) = "A"
    : C(X) = 14
6240  CD$(X) = TPCARD$ + GOBACK$ + CHR$ (1
        49) + RIGHT$ (C$(X),1) + S$(X) +
        CHR$ (170) + GOBACK$ + BOTCARD$
6245  IF X / 6 = INT (X / 6) THEN GOSUB 62
        70
6250  NEXT
6255  FOR X = 1 TO 52
    : CH(X) = 0
    : NEXT
6260  RETURN
6270  REM   ANTE
6275  IF N = - 1 THEN PRINT
    : PRINT "PLEASE ANTE:"
    : PRINT
    : PRINT
    : N = 0
    : RETURN
6280  N = N + 1
    : IF N = 5 THEN 6320
6290  IF P(N) < > 0 THEN PRINT P$(N) + " I
        S BUSTED."
    : PRINT
    : RETURN
6300  R(N) = R(N) - 1
    : PRINT P$(N) + " IS IN."
    : PT = PT + 1
    : PRINT
    : RETURN
6320  PRINT "ARE YOU IN?"
6325  C$ = INKEY$
```

```
6330  IF C$ = "Y" OR C$ = "y" THEN R(5) =
        R(5) - 1
     : PT = PT + 1
     : RETURN
6340  IF C$ = "N" OR C$ = "n" THEN END ELS
        E6325
```

The substance of the shuffling subroutine is contained in lines 6060–6090. In 6060 we set up a loop to get us the 40 cards we'll be needing. In 6070 we generate a random number from 1 to 52. In line 6080 we encounter our CH(X) variable, which we originally DIM'd at 52 values. At this point, since no other value has been assigned to them, all 52 CH(X)s equal 0. So the first time through, regardless of the value of C(X), CH(C(X))=0. Processing therefore goes to 6090, where CH(C(X)) is set at a value of 1. We're trying here to make sure we don't get a number from 1 to 52 we already have, so the next time we generate a random C(X) at 6070, if it happens to be one we've already generated, i.e., one which already has a CH value of 1, we go back to 6070 to generate another number until we have one that isn't already taken. After we've successfully generated a random number from 1 to 52 that hasn't already been generated, we simply go on and generate the next one until we have all 40 we need.

Lines 6130–6230 translate the number from 1 to 52 into a card value. Line 6130 gives us S(X), the suit value of our card, in a deceptively complicated equation. C(X)-1 will give us a number from 0 to 51; the INT value of that number divided by 13 will give us a number from 0 to 3, and the +1 gives us an S(X) anywhere from 1 to 4. If you don't understand how this formula works theoretically, try plugging some values into C(X); you should get the hang of it that way. The point is, this gives us just the right suit values of our 52 cards, based on whether the given card falls into the first batch of 13, the second batch, the third batch, or the fourth batch. Line 6140 translates S(X) into a suit symbol, S$(X).

Line 6180 is another cute formula, this time to give us our true C(X) value. What we're doing here is decrementing C(X) by 13 if C(X) is greater than 13, and doing it again and again if necessary until C(X) is not greater than 13. This ultimately gives us a value of C(X) from 1 to 13; line 6230 will take care of the problem of C(X)=1 (later in the program, when figuring straights, we'll be switching this around again). So now we have C(X) values from 2 to 14, and by the time we finished looping through all this 40 times we have 40 different cards and we're ready to deal them out.

Line 6240 is another graphics statement, this time creating our physical cards on the screen, a process we'll discuss in depth in the next chapter. Then we come to line 6245, which is how we mask the slowness of this shuffling process. What we've done here is put in a counter that says that if X divided by 6 is an integer, a whole number rather than a whole number and a fraction, then we'll GOSUB to 6270. Looking down to these lines, you'll see now why we set N = −1 in 6010. This subsubroutine simply collects the antes from each player. If N = −1 (our starting point), then 6275 prints PLEASE ANTE on the screen, increments N to 0, and then RETURNs. Six X's later in the original loop we'll come back to 6270 again, and this time we'll skip past 6275 to 6280, incrementing N again, following the procedures in 6290–6300 for values of N less than 5, and 6320–6340 for when N does equal 5. You'll remember that P(N) refers to our player numbers from 1 to 5. If P(N)<>0, then that player is out of the game, hence the message delivered in 6290. If the player is still in the game, then his bankroll (R(N)) is decremented, the pot (PT) is incremented, and the appropriate message in 6300 is sent to the screen.

If N = 5, then we're talking about the human player, P(5), so we go down to 6320. Once again we use INKEY$, and assuming that our player opts to stay in the game, his bankroll too is decremented and the pot incremented. If he bows out, then we END the program. Although I have not done it myself, you could add a save-game feature not unlike what we used in ''Space Derelict!'' so that the player could return to the game with each player's bankroll entered into a disk file to be pulled out at a later playing. All you have to to do is save R(N) for N equal to 1 to 5 and then throw in a REPLAY SAVED GAME option at the outset of the program to replace line 50020 in the housekeeping section.

What have we done here? By interpolating this sub-subroutine of anteing into our shuffle subroutine, we have completely masked the fact that we've been shuffling the cards. As our player has sat at the computer he's been busy watching each player ante and then anteing himself. As far as the player is concerned, the only thing that separated one ante from the next was a short but suitable pause. But for us, we've won the battle of squeezing a longish routine into the program without hindering the playability of the program. This is an important technique that is well worth remembering in your own programming, but it has one significant drawback: It can get very complicated and confusing. In the case of shuffling the cards it was relatively simple because we had an easily adaptable subsubroutine, the anteing, waiting in the wings to volunteer as our decoy. But in many cases you won't have so willing a victim, and you may be setting yourself up for some hard programming work. But it's usually worth it in the long run, especially in a game program.

The last thing we do after our shuffling routine is completely looped is reset all our CH(X)s to 0 in line 6255. Now we can RETURN and get into a little graphics.

10 THE LOOK OF THE GAME

LOOKING GOOD IS THE BEST REVENGE

When you sit down at a computer terminal you face a tabula rasa—a blank screen waiting for the juice to get turned on. Flip the on button, and you see a bunch of dots on the screen. As a rule those dots are arranged into letters and numbers, but they can also be arranged in either abstract blocks or a preset collection of symbols. Much of the time we don't think very much about the arrangement of those dots; it never crossed our mind to worry about their placement in "Space Derelict!" because all we were interested in there was the scrolling of text from the bottom up, with an occasional CLS to start us anew. But there's a truism about computer screen displays that you might have heard from other quarters: Neatness counts. Especially as you get into complex game design you'll find yourself needing a firm understanding of how screen displays are formed and how you can reform them to your own specifications. Fortunately, your TRS graphics make this very simple; despite the fact that there is no simple built-in high-resolution facility, the graphics that are built in are as easy as pie, once you get the hang of them, and for simple uses such as card games and the like, they can't be beat.

Some of the differences from Model I through Model 4 are really going to start to show in this chapter, but 90 percent of what is said here is applicable to all this series of TRS machines. The big difference will be your lack of the special characters from CHR$(192) to CHR$(255), meaning you may not have access to the suit symbols and the little pointing finger. But you can always substitute text for the graphics without too much real loss. The first version of this game, written for an Apple II (a notoriously truculent machine in the graphics arena), used no graphics whatsoever and resorted to such usage as 2 HEART for 2♡. The game played just as well without the pretty little card pictures, but I must admit it is nice to have them on the Model 4. In any case, if you can't do exactly what we're doing in this chapter, you can come pretty close, and the information will nonetheless be useful to you. The essential point we'll be making is that an attractive screen display is a sine qua non in programming. If your display is a confusing mishmash, if it lacks important information that the user requires, if it looks really second-rate, you're going to deter your users from using it. People will take one look at your game and say "Amateur" and go on to something else. And since TRS graphics are so relatively simple, there's no real excuse for this. What we're going to discuss now should be helpful to you regardless of your machine.

Unless we do something to stop it, every time we send a PRINT statement from the computer to the screen, that PRINTing begins at the very left of the screen, right below the last line that was printed. If a CLS command has most recently been executed, then the PRINTing begins at the top left of the screen; if text already extends down to somewhere in the middle of the screen, then the PRINTing begins to the left directly below what's already there; if the screen is full, then the text is scrolled up one line: Our new data appear at the very bottom, and every other line moves up one, while the very top line disappears altogether. This is all very logical and in many situations quite desirable. But sometimes we don't want data to scroll up on the screen. Maybe we're asking our user to type his name in on a blank line on an imaginary form, or showing a game score that we always want to appear at the bottom right corner of the screen, or whatever. Then we have to take matters into our own hands.

The essential ingredient of TRS graphics is the @ function. Using @ allows us to print whatever we want to the screen wherever we want to print it. In the back of your manual you'll find a video display work sheet in which the screen is broken up into numerous small boxes on a grid (1920 of them on the Model 4). The screen is also broken down by rows and columns, 24 of the former and 80 of the latter. Note that like most things computer all the numbering begins with 0 rather than 1. On the Model 4 this is more an attribute than a detriment, however, because we can easily determine all the left columns because we know they'll be in increments of 80 (0, 80, 160, etc.). But regardless of which model you have, these numbers, the ones from 0 to 1919 (or less on non-4s), and the rows and columns, are our guideposts to printing on the TRS. Each of these locations, be they pure locations from 0 to 1919 or row/column locations (e.g., 0,0 or 10,20), is a point at which we can print a character, the point and character being totally at our discretion. The form of this powerful function is

```
PRINT @ X, "My name is legion."
```
or
```
PRINT @ (10,20), "Want to make something of it?"
```

PRINTing @ a location puts whatever follows the comma in the statement at that location. The information printed can be either text or characters, variable or constant.

The best way to begin preparing a screen display is to make a photostat of your video display work sheet and play around with it (if you ask me, the manuals should come with about 500 of these blanks for user doodling). Figure out your overall design and where you're going to want to see what. Make it both attractive to the eye and easy to follow. And save your work sheets because they'll help you fill in the numbers @ where they belong.

In most games there is some sort of playing field that remains static while the pieces keep moving. This concept is obvious in chess or checkers or backgammon, but it's also quite true of poker (and just about any other strategy game you'll be designing). In poker the field is our five players sitting around a table. Somehow we have to represent visually this gang of five on the screen; we have to know who they are and where they are sitting in relationship to each other. We have to know who is still in on a given hand and who may have dropped out. We want to know if a player has busted and left the game altogether (in which case we don't want to see him at all). In addition, we want to keep an eye on the pot so we'll know how much is at stake, and we also want to be able to see our own stack of chips so we'll know where we stand. We'll also want to see our own hand and at least the backs of the cards of the other players. And finally we'll want to know what the other players are doing, whether opening or passing or raising, and how many cards they are taking. Again, it's best to design your game first on paper to show where exactly everything will go on the screen; judicious and frequent use of PRINT @ will allow us to execute it.

Here's how we're going to be accomplishing all of that in ''Five-Card Draw.'' The screen will be divided into six sections, two rows of three sections each. The top left-hand section will be player number 1, Marvin; on his right will be player number 2, Gerry, and next to him will be player number 3, Walter. Directly below Marvin will be our only female card-sharp, Betsy. You (the player) will have the section directly below Gerry, and to the right of you will be a box of ongoing information, including the amount of the pot, the amount of the present bet, and so forth. At the top of each section we'll have the player's name, and at the bottom, a ''chat'' line, where we'll print the information about what the player is doing, be it a bet, or how many cards he is taking, or whatever. But before we start PRINTing @, let's do some character building.

BLOCK THAT BLOCK

We've already discussed the way the TRS uses the ASCII character set. Much of this use conforms to industry standard, but the Tandy people have very cleverly added more to it than just the standard alphanumeric characters. By toggling internal switches back and forth you have access to a complete set of varied block graphics, and, on some machines, special characters like suit symbols as well. Combining these block and special characters is a simple task once you know how, and it's no more complicated than any concatenation of strings. Let's go back and see how we've done it so far in the poker program.

The first thing you'll remember we did was turn the special character set on with a CHR$(21) in line 60025. This gave us access to the suit symbols and the other specials. We began putting this together in line 50120, where we created FINGER$ by concatenating CHR$(244) + CHR$(245) +

CHR$(246) (if some of the special characters seem meaningless to you as printed in the manuals, you might want to try printing them to the screen to get the real poop on them; they look pretty weird all blown up, whereas they look quite normal on the video monitor).

The next thing we did, beginning at 50140, was a little more complicated, making use of the little sets of blocks that reside from ASCII 128 to 191. Take a look at them in the manual as you follow this discussion. First, we took 151, 131, 131, and 171 and concatenated them; this gave us our TPCARD$, which is just that. Then we created MIDCARD$,which is 149 + 128 + 128 + 170—the left and right sides of our rectangular card plus two empty spaces in the middle—and BOTCARD$, which is 181 + 176 + 176 + 186, the bottom of the card. GOBACK$ is a tricky one. We used CHR$(26), which if you look it up is one of a number of cursor moves available to us around this neighborhood of ASCIIs. All of these cursor moves, just like the reverse video display or the turning of the blinking cursor on and off, are—in a manner of speaking—printable. When we PRINT CHR$(26) we move the cursor down one block from its present position; we could have moved it up, back, forward, to the start of the next line, or just about anywhere. What we do in creating GOBACK$ is move the cursor down from its last position and then four spaces to the left, each one of which is ASCII 24, and which we address in our STRING$(4,24) function in 50155. Finally we created the actual blank cards by concatenating all these pieces, along with the cursor moves, in line 50160, calling it all ART$—TPCARD$ + GOBACK$ + MIDCARD$ + GOBACK$ + BOTCARD$. Farther along the line, at 6240, we created the other sides of the cards, the ones with the characters on them, in much the same way. The only difference there is the inclusion of C$(X) and S$(X) in place of the two blank 128s in MIDCARD$.

All of this gives us nice little blocks of graphics three rows high and four columns wide, and they can be treated as if they are only one graphic character, the string ART$. When we PRINT @ X, ART$, the whole kit and caboodle is displayed onto the screen, which is not a bad deal at all. Character graphics like this are terribly easy to create, and with a little experimentation you can come up with just about any simple graphics display you might need. The only piece we've left out so far is ERAS$ at 50161–50162. Here we're creating a three-by-four block of nothing but spaces; we're going to use this as an "eraser" to delete any cards we want to take off the screen when a player throws in all or part of his hand. In other words, if we want a blank card we PRINT ART$, if we want the face of the card we PRINT CD$(X), and if we want to delete a card we PRINT ERAS$. And we print them where we want them by using an @. But there's still one fly in the ointment, especially when you incorporate cursor moves into your character blocks, and that is the cursor itself. All the while you're setting up nifty little graphic displays, the cursor is lurking behind the screen waiting to act in seemingly unpredictable ways to wreak havoc on what you've been doing. Actually these ways are not unpredictable at all, but the important concern for us is that the cursor can be sent packing entirely, provided we remember one very important point. In using PRINT @, especially when combined with character blocks but as a rule all the time, *always suppress the cursor with a semicolon.* If you follow a PRINT com-

mand with a carriage return, either by starting a new program line or with a colon for multiple statements on one line, the screen will automatically place the cursor flush left of the next line, thence following its normal scroll protocol, which could send your whole beautifully designed screen display up and away into neverland. However, if you follow your PRINTs with a semicolon, as in
PRINT "Long live Babar.";
or
PRINT @ LOK, "Long live Queen Celeste.";
or
PRINT "Where's that";:PRINT "rascal Arthur?";
or, in the case of INPUT statements
INPUT; X
or
INPUT; X: IF X = 4 THEN PRINT "Most of the Marx Brothers.";
the cursor will stay right where it ended up, and so will your screen display, which is exactly what we want.

So now let's get down to the business at hand.

THE GRAPHICS ROUTINES

At this point in the poker program all the players have anted, and the cards have been shuffled. Now it's time to deal out the hands.

"Five-Card Draw"

```
18000 REM  DEAL
18010 CLS
18050 IF P(1) = 0 THEN PRINT P$(1)
18070 IF P(2) = 0 THEN PRINT @25,P$(2)
18090 IF P(3) = 0 THEN PRINT @50,P$(3)
18110 IF P(4) = 0 THEN PRINT @960,P$(4)
18130 PRINT @985,P$(5)
18140 FOR X = 1 TO 5
18150 LOK = (80 * X) + ((X - 1) * 3)
18160 FOR Y = 1 TO 5
18170 IF Y > 1 THEN LOK = LOK + 20
    : IF Y = 4 THEN LOK = LOK + 900
18180 IF P(Y) < > 0 THEN 18200
18190 IF Y < > 5 THEN PRINT @LOK,ART$;
18195 IF Y = 5 THEN PRINT @LOK,CD$(20 + X)
    ;
18200 NEXT Y
18210 NEXT X
18240 GOSUB 31000
    : RETURN
```

Lines 18000–18240 contain almost the entire sum of our graphics layout. This is how we get the cards onto the screen, and this is how we make sure the human player can understand what we're doing. And here too we encounter our first @s.

The first thing we do in this subroutine is print to the screen the names of all the players who are still in the game (lines 18050–18130). First we verify that P(X)=0, i.e., the player is still with us in the hand, and then we print @ a certain number the name of that player. The best way to follow this discussion now is to look at your video display sheet. We'll be printing P$(1), MARVIN, @ 0; since we've just CLSed, this is the automatic default in line 18050. Now look over to column 25, row 0. This could be expressed either as (0,25) or, as we've done it, simply 25 (line 18080). Line 18090 prints P$(3) @ 50 (or 0,50), while we skip way down to 960 (or 12,0) for the printing of BETSY, P$(4). P$(5) is printed @ 985; no conditional is used here because it is impossible for the human player not to be in at this point.

So now we've printed the names of each player across the screen. Our use of @ was not particularly complicated, nor will it become complicated as we progress. The next thing we have to do is print the graphic cards. Needless to say, we will not display to the human the values held in the other players' hands, hence all we have to do is display the blank ART$ cards on the screen. We'll lay them out in a diagonal (it looks good that way) right below the names of each player. The only card values we'll actually see will be the human player's.

The thing to keep in mind here is the relationship of the immediately adjacent points to our most recent point. For argument's sake, let's say that we start with point 100. If we wanted to print at the point directly above 100, we would subtract 80 to do so—i.e., 100−80=20, the point directly above point 80. Similarly, we add 80 to our value, 100, to get 180, the point directly below point 100. Moving left or right is a simple matter of subtracting 1 or adding 1, while moving in an up-right diagonal is a −79, an up-left diagonal a −81, and so forth. If you have a smaller display, simply substitute the appropriate number, e.g., 64 or 40, for the 80. It all comes out the same in the long run.

It doesn't take long to get a feel for the point relationships on the screen. And once you have that feel, you can place anything you want from single characters to strings to block graphics. 18140–18210, a mere eight lines, is how we display 25 cards, in regular order of deal (one at a time for each player), some face up and some face down, depending on which player is receiving them. And the formula is not complicated, while the display itself is quite full and detailed. If we can deal five hands of cards this compactly, you can do similar manipulations with similar precision.

In 18140 we set up our loop of five cards, one for each player, and then in 18150 establish the LOK variable, as in LOKation (LOC is already taken by the TRS as a reserved word for establishing the record number in a file), for each card. The formula in 18150 is simple, using the basic 80-column relationship of points, throwing in a couple of extra points for spice. Since our ART$ is four characters long and we're laying our cards one on top of the other in a diagonal, we'll want to move over to the right three points and down one point so that the cards ''overlap.'' (In reality an overlap replaces the last character with the present character.) So the 80 in the formula is the easy part, multiplied by X to put us at the right horizontal position, while the right half merely figures, on the basis of which card we're up to, how many points to move to the right, or the vertical position.

Next we loop through the players, the Y value in 18160. We move LOK to its appropriate position under the player of choice in 18170 and then test to see if the player is in in 18180. We'll want to move LOK in either case, which is why this test wasn't performed sooner; our value of LOK depends on manipulating all five players, as you can see from the way the formula is designed. It's in 18190–18195 that we finally do our PRINTing @ LOK, selecting either the blank ART$ cards or a real card depending on who we're dealing to. As you would expect, the human player gets CD$(21 through 25). The important thing to note is the semicolon ending both these lines. We most definitely do not want to issue a carriage return, or Enter, after this printing. We could erase all or part of what we've already drawn if we were to do that, and as I said before, but which nonetheless bears repeating, you should end all PRINT @s similarly, with the semicolon carriage return suppress. If you don't, you might get the results you wanted, but I wouldn't count on it. That cursor is just sitting there, waiting to tear asunder everything you've just done.

ONGOING INFORMATION

When you sit down to play a hand of ''Five-Card Draw'' with human opponents, there are a number of givens in the situation that you seldom pay close conscious attention to. For instance, you don't regularly analyze the exact size of the pot: either there are a lot of chips in it, or there aren't; you know at a glance what sort of money is at stake at any point in the game. Similarly, the size of your own stack of chips gives you a general idea of how you're doing without you're having to add them up every two seconds. In our computer game, however, we don't have these visual reminders, so we have to invent them.

You can be almost certain that just about any strategy game you design will need some area of the screen devoted to an ongoing tally of the game, or at least an updated score after each round of play. The important thing for you to decide is how much ongoing information is necessary to the player, and then where to put it. In ''Five-Card Draw'' we had the bottom right-hand corner of the screen nicely left over for us, so that's where our information will be going. That's the easy decision; what to put there is a little harder.

First we want to know who the dealer is the deal will pass (of course, from player to player after each hand). We'll also want to know how much money is in the pot and how much money we have ourselves. Since it is not particularly crucial at all times to know how much money the other four players have, this information is left out. If the hand has been opened by one of the players, we'll want a reminder of which player that was. Also, we'll want to know how much the bet is. If the opener bets one dollar and everyone sees that bet, no problem, but if the opener bets 2, the next player raises 3, the bet to you is 5, the next player raises 3, etc., the bet comes around to you again as 3—it's usually not hard to keep track of this in a real game, but the fact that real money is not changing hands here tends to complicate things for us. Hence the constant reminder. And finally, we'll want to keep track of the number of raises because of the three-raise limit.

This is all very simple, and all it requires is that we print the appropriate information at the appropriate LOK locations. The 80 relationship of all these LOKs should be quite obvious to you.

FIVE-CARD DRAW

```
31000 REM    SET ONGOING INFORMATION
31010 IF STARTER < > 0 THEN PRINT @1090,P$
         (STARTER) + B$(7);
31030 PRINT @1170,"DEALER: " + P$(D);
31040 PRINT @1250,B$(12);
31050 PRINT @1250,B$(4);BE - LB(OP);
31080 PRINT @1330,"NUMBER OF RAISES =";NB;

31110 PRINT @1410,B$(5);PT;
31120 PRINT @1490,B$(12);
31130 PRINT @1490,B$(6);R(5);
31140 RETURN
```

The only problem with understanding this routine is that it introduces a number of new variables, but we're going to have to digest them sooner or later, so we might as well get it out of the way now.

Line 31010 takes us right to the first one, STARTER. STARTER, as it turns out, refers to the player from 1 to 5 who makes the opening bet. So if STARTER = 1 then player number 1, Marvin, was the opener; if STARTER = 2 it was Gerry, and so forth. If STARTER = 0 then nobody has opened yet, so we skip this line entirely. P$(X), you'll remember, was the name of player X, so P$(STARTER) will give us the name of the opener. B$(7) = ": Opener".

From this point on you'll occasionally see a clearing of each line (the printing of B$(12)) before the line is printed. This is to make sure we don't leave any old information behind us. For instance, let's say the bet to Walter is $11. He throws in his chips, and then the bet to Betsy is $7. Between Walter's bet and Betsy's we'll have cleared this part of the screen so that we'll print the appropriate $7 when the time comes. What we don't want is the leftover second digit of Walter's bet, turning our $7 into $71 (some of this might make more sense if you look at it on your screen rather than trying to read about it; it's all very clear when it's right there in front of you). It would have been unnecessary to clear the STARTER line this same way, since that line will always be empty to begin with and will never be replaced during a hand: There's only one opener in any hand of poker.

Line 31030 gives us the dealer (and again we have no need to clear—there's only one dealer per hand). We've already met the D variable, which we originally set at 1 in our housekeeping routine. Line 31050 opens up a whole new can of worms, which I'm going to delay for a while. Suffice it to say the BE-LB(OP) equals the amount of the bet to a given player. This value will be accurate regardless of which player's turn it is. BE-LB(OP) comes from the betting subroutine of the program, which for me proved to be among the most difficult, probably because there were so many things going on at the same time to keep track of. We'll save all that for the next chapter.

Line 31080 tells us the number of raises, which obviously is the variable NB (as in Number of Bet). PT is the pot, and R(5) is the amount of money in the human player's bankroll, which brings us to the end of this subroutine.

Now we've put everything we need on the screen. The player sitting at his computer will see visual representations of five players including himself, as well as a running tally of how he's doing and what's going on in the game. Next we need to set up subroutines for the visual progress of the game itself.

LOK ON THE MARCH

Remembering back to our discussion of the screen layout, you'll recall some room set aside for "chat." It is on this part of the screen that each player tells the human what he is doing, be it betting, taking cards, dropping out, whatever. Each player, including the human one, has his own chat line. In the case of the human, this is where the computer will be requesting player inputs and, occasionally, sending a message or two back. The chat lines are located directly below the individual card displays.

We don't have to do much in this subroutine other than establish our LOK for each chat line and then clear that particular chat line space with a B$(12). The programming looks like this:

FIVE-CARD DRAW

```
7000  LOK = 720
    : PRINT @LOK,B$(12)
    : RETURN
7040  LOK = 745
    : PRINT @LOK,B$(12)
    : RETURN
7080  LOK = 770
    : PRINT @LOK,B$(12)
    : RETURN
7120  LOK = 1680
    : PRINT @LOK,B$(12)
    : RETURN
7160  LOK = 1705
    : PRINT @LOK,B$(12)
    : RETURN
```

All we do is set the LOKs and clear the lines. The routine can be accessed for any individual player however we need it, simply by GOSUBing the correct lines (GOSUB 7160 for the human player, for instance, or GOSUB 7080 for player 3, Walter). This whole subroutine is placed fairly low in the flow of the program because we use it quite frequently, and there's no point in slowing ourselves down unnecessarily.

Whereas the above routine will be used on a per-player basis whenever we need it, every now and then in the game we'll want to clear everybody's chat lines all at once. For instance, after all the players have completed the first round of betting, each is going to ask for a certain number of cards. We don't want to be looking at a screen where one player says "I'LL SEE THAT" while another player is saying "I'LL TAKE 2 CARDS." That's a sort of visual cacophony similar to all the players at the table talking at once. So subroutine 19000 accesses all the chat sections, clearing them one at a time so the next part of the game can begin.

```
19000 REM  CLEAR SCREEN A BIT FOR TIDINESS

19010 GOSUB 63000
    : GOSUB 63000
    : GOSUB 63000
19020 GOSUB 7000
19030 GOSUB 7040
19040 GOSUB 7080
19050 GOSUB 7120
19060 GOSUB 7160
19070 NB = 0
    : BE = 0
    : FOR I = 1 TO 5
    : LB(I) = 0
    : NEXT
    : GOSUB 31000
    : GOSUB 63000
    : RETURN
```

An easy one, that, clicking off the accesses one by one, starting with Marvin and ending up with the human. Even though we haven't said explicitly what some of those variables are in 19070, as you can see, they are all cleared to 0 prior to another access of the 31000 info display routine. Since this subroutine occurs only at landmark junctions in the play of the game, the information rendered in those variables has become worthless, so we get rid of it. However, we will still see all the rest of the info display information, with zero values for the information represented by those emptied variables.

As a point of information, I could have squeezed all the data from 19010 through 19060 onto one line, but there comes a point when condensed programming for speed simply becomes obsessive behavior. If I were pressed for space, however, I certainly would have done it.

By the way, you've probably noticed the arithmetic relationship of the lines accessed in this routine: 7000, 7040, 7080, 7120, and 7160. Unfortunately, in TRS Basic we cannot use a routine such as:

```
FOR X = 7000 TO 7160 STEP 40
GOSUB X
NEXT
```

The temptation is strong, but Basic does not recognize variables in GOTO and GOSUB statements. Sorry about that.

DROPPING OUT

There are a number of little sidedish routines in the poker program also related to screen formatting that we won't bother talking about now. They're small and easily digested, tiny variations on what we've been doing up to this point. Aside from these, two more major screen events and one minor one remain to be discussed. The first of these concerns when a player drops out of the game, and the second concerns when the hand is over and all the players show their cards. The third, minor one points to the winner of the hand.

As a hand progresses, it is not unusual for a player to drop out, either because his cards are no good or because the betting has gotten too steep for him. When this happens we want to remove that player's cards from the screen. We most pointedly do not want to show what those cards are if they belong to one of the computer-operated opponents. Just as in real-life poker, when a player drops out, his hand is discarded face down and no one ever finds out what he was holding. On the video screen we want to represent this by deleting either the backs of the cards, in the case of a computer opponent, or the faces of the human's cards, depending on who is dropping out. It is perfectly acceptable for the human to drop out of any hand at any time; the hand will progress without him, just as in real poker, and he will see that hand progress as a spectator. He will rejoin the game at the next deal.

There's nothing complicated about this subroutine, which proceeds from lines 40000–40060.

FIVE-CARD DRAW

```
40000 REM  CHANGE DISPLAY WHEN A PLAYER FO
         LDS
40010 LOK = 412
    : IF OP = 2 THEN LOK = 432ELSE IF OP
         = 3 THEN LOK = 452ELSE IF OP = 4
         THEN LOK = 1372ELSE IF OP = 5
         THEN LOK = 1392
40020 FOR I = 1 TO 5
40030 PRINT @LOK,ERAS$;
40040 LOK = LOK - 83
40050 NEXT
40060 RETURN
```

What we're doing here is overlaying our blank block, ERAS$, over the extant blocks, be they ART$s or CD$(X)s. We establish our LOKs backward from the last card dealt this particular player to the first, establishing that backward starting point in line 40010. The variable OP once again represents any one of our players from 1 to 5 (the exact derivation of OP need not concern us now). Once we have our LOK, we establish our five-value I loop and then print ERAS$ over the appropriate blocks. Note again our semicolon at the end of 40130. Since each block is 83 points away from the next one in a given hand, 83 is the value adjustment to LOK in line

40040. When this subroutine is run, all the other hands remain exactly as they are.

The subroutine for displaying the hands is similar to our original deal. At any point in the game, the players are holding these cards:
Player 1—CD$(1 through 5)
Player 2—CD$(6 through 10)
Player 3—CD$(11 through 15)
Player 4—CD$(16 through 20)
Player 5—CD$(21 through 25)
This is quite logical, and if you keep it in mind the hand-display routine will make total sense.

FIVE-CARD DRAW

```
47000 REM  SHOW CARDS
47010 FOR I = 1 TO 5
47020 LOK = (80 * I) + ((I - 1) * 3)
47030 FOR Y = 1 TO 5
47040 IF Y > 1 THEN LOK = LOK + 20
      : IF Y = 4 THEN LOK = LOK + 900
47050 IF P(Y) < > 0 THEN 47070
47060 PRINT @LOK,CD$(((Y - 1) * 5) + I);
47070 NEXT Y
47080 NEXT I
47090 RETURN
```

This time we'll display each player's hand all at one time. I is the counter for the number of the card, and Y is the counter for the player number. LOK is established as it was before, and the actual printing of the correct CD$ is determined in line 47160. (Y − 1)*5 gives us a value from 0 to 4 multiplied by 5, to which we add the appropriate I value; as we progress from player 1 to player 4, this will give us CD$(X) from CD$(1) to CD$(20). The net result of this is the overlaying of the correct CD$(X)s over the previous ART$s.

All the last minor routine does is figure the LOKs at the point at which we've printed the names of the players:

FIVE-CARD DRAW

```
46100 LOK = 0
      : RETURN
46200 LOK = 25
      : RETURN
46300 LOK = 50
      : RETURN
46400 LOK = 960
      : RETURN
46500 LOK = 985
      : RETURN
```

When a player wins we're going to print a FINGER$ next to his name. These minisubroutines get us to the right place.

As I said, this just about wraps up screen formatting, which ultimately boils down to a relatively simple business, as programming goes. The most important thing to keep in mind is to make the screen as attractive as possible. A good look enhances a game; a bad look kicks it in the pants.

11 THE MEAT OF THE GAME

Early in this book we explained every little thing no matter how insignificant. Now there are fewer explanations, on the assumption that you have at this point absorbed a lot of programming lore and can see the logic in complex statements that might have been unclear earlier on. We can pull away now and take a broader look at what we're doing. You've become a Basic literate: We've covered just about every command, and we've been through so many trenches now that you really do know what the battle is all about. You've learned quite a bit (and so have I). This points up one of my most fervent beliefs about programming: The best way to learn it is to do it, and studying other complete, understandable programs is almost as good as creating them yourself. Unfortunately, most good programs are kept under lock and key; the manufacturers copy-protect them, and we never get to see how the state-of-the-art sophisticates are really doing it (although, to be honest, most purchased programs are written in machine language). But we do have the chance in books (sometimes) and in magazines (often) to strip away this cloak of secrecy. After investing the roughly $2000 you plunked down to obtain your TRS-80, you really should take the next step and get yourself a few $20 magazine subscriptions. *Micro 80* is just about a must for Tandyites, covering almost all the TRS-80 incarnations (the dialects of which are sometimes more similar than you'd expect) with special articles devoted to individual machines. Included are everything from games to assembly language tutorials to Basic programs to run your farming operation. Then, of course, there are the general magazines such as *Byte, Personal Computing, Creative Computing,* and so forth. These vary in intensity, and the rule of thumb I follow is to buy them on the newsstands if they have articles of interest to me, which is about 25 percent of the time for any one of them. Here, of course, it all depends on what you're interested in. There is a lot of information out there, and it seems to change all the time; magazines are your best connection with what's what with your computer.

From here on this book will still have plenty of information, including new tricks on some of the old commands we've already been playing with, but the concentration will be more on structure and design and logic than on the actual programming. If you were to stop reading this book right now, you would already know enough to write some pretty good original programs on your own. But stick with us and you'll learn a little more.

This chapter will include all the rest of the poker program. It will concern itself with the reading of the cards by the computer, the creation of the four computer players, and the process of betting. It will cover both the first round when the cards are originally dealt, plus the second round of drawing new cards and betting some more. Finally, it will show how the winner is decided. Needless to say, that means a hell of a lot of programming and a very long chapter. I had to decide whether to bunch all the similar routines together, or to follow the flow of the game. Because at this point the most important thing to keep in mind is design structure, I have gone with the latter approach, which means we'll be bouncing around a little to different sorts of routines rather than concentrating on the related routines within the program, but by following the flow we'll be able to keep in mind more clearly how the program works overall. Since you now have a good foundation in how Basic works, this is the more logical approach. And first we'll need to get a good idea of the "personalities" of the computer-manipulated players.

MEET YOUR OPPONENTS

The five-card-draw program in this book pits the human player against four computer-controlled players named Marvin, Gerry, Walter, and Betsy. You'll learn more about these four opponents when you see how they're programmed, but you might wonder right off why these four, and why these four styles of play? It's very simple: These four people really exist—almost. They are, in fact, my real poker buddies, with some slight variations for the sake of the game—most noticeably Betsy, who is in real life named David (I felt that we needed some female presence in this book; sorry, David).

This use of human models made my job a bit easier. And I like to think it was a good idea worth propagating. The real-life models gave me a starting point and a focus. And now when I play this poker program I can really imagine my friends playing the way their programmed counterparts do (although lately I've been noticing the divergence that I programmed in for the sake of a better computer game). I didn't use my friends 100 percent, but they were certainly the jumping-off point, and some came in closer than others.

As I said, you'll learn more about these four when we get into the game, but I thought I'd introduce them now just to flesh them out a bit. Marvin is the most casual of the four. His philosophy of poker is that it's a fun night out with the boys, and whatever he loses is simply the cost of an evening's entertainment. He also has a philosophy—"That's why they call it gambling"—when he goes for a particularly unlikely draw, such as two cards for an inside straight. People like Marvin make gambling casinos the profitable ventures they are, because Marvin, who is in all other respects a man of awesome intelligence, is a terrible card player. And in our program, Marvin is the easiest opponent to beat, but he's so unpredictable that he's hard to pin down. You'll see why when you sit down against him.

Betsy is one step up from Marvin, by virtue of being somewhat of a student of the game and trying a few things that Marvin would never think of. Betsy takes most of the same unlikely chances as Marvin, but she's a bit more careful. She also firmly believes in bluffing and does so at the drop of a hat, usually at the cost of a lot of chips. The problem is, she can't always be bluffing, and how do you tell if she's really got something?

Walter is on the other side of average. What Walter tends to do that keeps him from hitting the heights is, as they say in the game, betting on the come—staking your money on cards you haven't gotten yet. An instance of this would be making a substantial raise on the first round of betting when you've got a four flush. It's a risky business, this betting on the come. Walter is a more reasonable bluffer than Betsy, and Walter tends to keep a kicker now and then. Walter is probably the sneakiest player of the four.

Gerry is the closest we'll come to a top-drawer player. In the real world my Gerry is not bad, but he's not as good as I've made him here. This Gerry is good at money management: He doesn't bet on the come, he's a conservative bluffer, he knows when and how to draw. The problem is that he plays a fairly dull game, although he does tend to do quite well over the long term. When you play this program you'll find that Gerry is usually the hardest player to beat.

So what's been created in the poker program is an approximation of these four players. But whereas at a real table on Friday night these flesh-and-blood characters are sometimes quite hard to pin down—sometimes Walter plays a better game than Gerry, sometimes even Marvin has a few surprises up his sleeve—in our program they have been minutely quantified. For the human player sitting at the computer, they represent four different levels of skill, as well as four different players, ranging from Marvin to Gerry (although the human won't know this hierarchy of skill until he's played the game for a while).

Once again, just to get it set in your mind, let's redefine the players:
P$(1) = Marvin (or P(1))
P$(2) = Gerry (P(2))
P$(3) = Walter (P(3))
P$(4) = Betsy P(4))
From this point on I'll be using their names and numbers interchangeably. This anthropomorphization of the program was helpful to me when I wrote it, and now I simply can't shake it. As I said, you might find that a similar anthropomorphization might help you as well in your own strategy games.

THE MEGALOOP

At this point in the running of the program the human player has anted and watched the cards being dealt. He sees before him his own hand, plus the backs of the cards of the other four players. The player will now study his hand and decide what to bet; likewise, the computer will study *its* hands and decide what to bet. The difference between the human and the computer processes for performing these identical chores is the stuff at the core of understanding how human intelligence differs from computer processing. The way a human thinks allows him or her to look at the hand and analyze it almost instantly; the human is capable of recognizing the various abstract patterns that comprise a hand of poker and classifying them immediately. The human mind does not have to run through all the possibilities from straight flush on down to know what he has; he simply looks at it and recognizes it—and no one really knows how. If we did, we could replicate the process with computers, but since we don't, we're forced to do it another, more tedious way: We must take every possibility and eliminate each and every one, or as they say in the trade, we have to do a lot of number crunching (or in this case, card crunching). Fortunately, the computer is pretty adept at number crunching and does it fairly swiftly, but in no way is this number crunching as swift as the human process of abstract-pattern recognition. It just happens to be swifter than human number crunching, which is why they invented computers in the first place, to perform tedious numerical chores.

The problem we face in this particular crunch is that, despite the speed of the computer, because there's so much crunching to be done this process is noticeably slow, as would have been the shuffling routine if we hadn't sneaked around it. But in this case we don't have a convenient distraction to misdirect the eye of the human player; this time we can't be magicians distracting with our right hands while our left hands are fishing around in our pockets for rabbits. So all the processing contained from line 90 to line 1970, which is quite a lot, will transpire with nothing else going on except the human staring at his hand; our human player will notice a definite pause of 20 seconds or so while he waits for something to happen. Compounding this is the fact that 90 through 1970 is actually a FOR . . . NEXT loop of humongous dimensions: Line 90 reads FOR X = 1 TO 5, which means that we must go through these lines five times, once for each player to figure out what all the hands are. This unfortunate, undisguisable delay is encountered sooner or later in all strategy games of any complexity, and it is just something you have to get used to (with some chess programs at high play levels there's a 24-hour processing delay between moves!). The best you can do is streamline the processing as much as possible to make it as swift as you can.

This programming from 90 to 1970 does a lot of things, but the gist of it is the doping out of what each of the five players is holding. After this section is completed, there will be a phase of betting by all the players. At one point when the program was finished I did try moving this around a bit: Instead of figuring all the hands at once for a 20-second delay, I tried figuring for one player, then having that player bet, doing it for the next player, then having him bet, and so forth, to see whether five shorter delays were more palatable than one long one. Ultimately, I decided they weren't. This is probably a question of personal taste, so if this kind of option arises in one of your games, I suggest you try it both ways.

The major concern I had in designing this section was the logic of the flow of figuring the hands. I could, of course, have simply laid out a program where the various cards in each player's hand were tried against each and every possible combination of cards, and going with the highest combination, but this would have meant trying some possibilities that could easily be eliminated. For instance, if a player does not hold one pair, there is no way in creation he can hold two pairs. So if we've eliminated one pair we can also eliminate two pairs. Furthermore, if we've eliminated the possibility that any two of his cards match, we can also eliminate three of a kind, four of a kind, and a full house. This kind of logic is simple, of course, but we take it to its fullest extreme, as you'll be seeing shortly. Our chief concern must be not wasting time on the impossible, and these lines of the program come as close to it as I could manage. As I look back on it I see here and there where another arrangement could have sufficed just as well, but the main idea would have remained the same—the least amount of programming to do the most amount of work the fastest. In strategy games where crunching time is of the essence, this must be one of our strongest guiding principles. Let's look at a piece of this part of the program now, taking it one step at a time to keep it manageable.

FIVE-CARD DRAW

```
90     FOR X = 1 TO 5
100    IF P(X) < > 0 THEN 1550
130    FOR Y = 1 TO 5
140    V(X,Y) = C(Y + ((X - 1) * 5))
     : V$(X,Y) = C$(Y + ((X - 1) * 5))
     : ST(X,Y) = S(Y + ((X - 1) * 5))
     : ST$(X,Y) = S$(Y + ((X - 1) * 5))
170    NEXT
200    REM  ONE PAIR
210    FOR N = 1 TO 4
220    FOR I = 2 TO 5
240    IF N < > I AND V(X,N) = V(X,I) THEN
         H$(X) = "P" + RIGHT$ ( STR$ (N),1)
         + RIGHT$ ( STR$ (I),1)
     : N = 4
     : I = 5
250    NEXT I,N
260    IF LEFT$ (H$(X),1) < > "P" THEN 610
270    IF V(X, VAL ( MID$ (H$(X),2,1))) >
         = 11 THEN H$(X) = H$(X) + "J"
280    IF MID$ (H$(X),2,1) = "3" THEN 390
290    IF MID$ (H$(X),2,1) = "4" AND
         RIGHT$ (H$(X),1) < > "J" THEN 610
300    IF MID$ (H$(X),2,1) = "4" THEN 1360
```

Line 90 will come as no surprise, since we already mentioned it. You may be wondering, however, why we bother to analyze five hands, since the computer is playing only four of them. The reason is that the computer will want to know what the human player is holding, not to adjust its own play, but to know whether the human has an opening hand of jacks or better. This will prevent the human from cheating later on. Line 100, which takes us to 1550, is our eliminator. At this point in the game any number of players (including the human) could be out of the game; we want to make sure we don't waste time figuring them too. Line 1550 is the NEXT statement complementing the FOR in line 90—a real megaloop no matter how you slice it.

The loop 130–170 does something very important, which we mentioned earlier in our expanded discussion of arrays. Throughout the program, we want to use the same variables to denote the cards for each player. Up to this point we know that player 1 has the first five cards dealt. He has C(1 through 5), C$(1 through 5), S(1 through 5), S$(1 through 5), and CD$(1 through 5). Player 2 has 6 through 10, player 3 has 11 through 15, player 4 has 16 through 20, and player 5, the human, has 21 through 25. The thing is, in figuring the hands we can't make heads or tails of these C's, S's, and so forth because we want to use our variables horizontally and not vertically. That is, we're concerned with each player's cards from 1 to 5 rather than all the players' cards from 1 to 25. So we create new variables to help us with this. V will be the same as C, V$ will be the same as C$, S and S$ will be ST and ST$ (we don't have to worry about CD$, which we analyze vertically). The formulas in line 140 turn C(1) into V(1,1), C(2) into V(1,2), C(3) into V(1,3), and so on. C(6), player 2's first card, will become V(2,1). By the same token, skipping down a bit, C(19) will become V(4,4). We'll always know which card we're talking about in which player's hand, and it won't matter which player it happens to be. We'll be using the X counter throughout the first part of the array—we'll never throw in a constant—but we'll be maneuvering the Y values from 1 to 5. The formulas on the other side of the equals signs in 140 do this translating for us; they are complicated and, sad to say, mathematically self-explanatory. There's not much of this business in this particular program, but you can encounter this sort of stuff (and probably even more of it and more complicated) in any strategy game. It's basic algebra; just try to remember what they taught you in ninth grade. Finally, the reason we didn't just create V and ST in the beginning, skipping the C and S family entirely, was for more speed in the shuffling routine, which this would have slowed down even further.

The best way to get a feel for these new variables is to look at them in action. The first thing we do in our elimination of possibilities for the hands is see if our player has a pair. If he does, there are numerous possibilities; if he doesn't, there are numerous other possibilities. This determination will send us on the right track right away. In 210–220 we set two nested loops to measure the V, or numeric value, of each of the cards in the hand and look for a match. We want to measure card 1—V(X,1)—against cards 2 (V(X,2)) through 5 (V(X,5)). So we'll start with N=1 in the outer loop against 2 through 5 in the inner loop. Then we'll measure card 2 against cards 3 through 5 (we'll have already measured it against card 1, hence the truncated looping in 210–220 of 1 to 4 and 2 to 5), card 3 against cards 4 and 5, and card 4 against card 5. Line 240 does the work for us, making sure that N <> I, then seeing if V(X,N) = V(X,I). If it does, then we do have a pair, so we now can give this hand an H$(X) value. H$(X) will be the name of a hand. At this point in the game we give artificial names that only we and the program will understand; in the case of a pair we will call it "Pxx". The "P" tells us we have a pair, and the "xx"—in this case the

string values of N and I—tells us which cards of the five are the pair. Let's say that the hand is this:
JACK CLUB
QUEEN SPADE
JACK DIAMOND
2 HEART
5 SPADE
In this case we would now have an H$(X) of "P13". Knowing what H$(X) is in this fashion allows us to perform a multitude of further calculations based on this information, as you'll see in a second. We use the RIGHT$ to eliminate the space at the left, as we did in "Space Derelict!" The remainder of line 240 simply augments the values of N and I so that the loop will be discontinued when we get to 250—we don't want to keep looping after we've found out what we want to know. And the format of line 250 shows how we can combine the NEXTing of two nested loops in one line, inner loop first, then outer loop.

So now we have either an H$(X) of "Pxx" or of "", an empty string. Line 260 sends us to line 610 if we don't have a pair; 610 will begin figuring whether this hand contains a straight or a flush, or parts thereof. Otherwise we go to 270. Here we're going to determine whether we have a high pair or a low pair. At this point we have H$(X) = "Pxx". The left side of the IF equation in 270 pulls out the value of the "x" in the middle—aka VAL (MID$(H$(X),2,1))—and tells us whether V(X,x) is greater than or equal to 11, i.e., jack through ace. If so, we amend H$(X) to "PxxJ" by adding "J" to H$(X). Now we know if we have an opener, jacks or better.

Line 280 is another exclusion like 260, and this time quite subtle. We know that we have five cards and that when we analyzed them we measured the pairs starting with card 1 against card 2 and so forth. This means that if the first card in our pair was card 3, we know that neither card 1 nor card 2 is equal to one another, nor to any other card of the five. In other words, we have either "P34" or "P35" and no chance of two pairs. So we won't bother to figure two pairs, and we'll go straight to 390 to figure three of a kind. Again, this saves us wasting time figuring an impossible combination. Line 290 does more of this. If our hand is "P45" we know we have a low pair and no other matching possibilities, so we scoot out to 610, which, again, is where we figure straights and flushes (we will be throwing away small pairs in favor of four flushes and four straights; i.e., in the hand:
9 DIAMOND
2 CLUB
9 CLUB
4 CLUB
ACE CLUB
we will discard the 9 of diamonds in hopes of the flush improvement). Note that if we have "P45J" we do not go to 610—we won't be throwing away an opener. If we do have this hand, line 300 takes care of it for us (the logical exclusion from 290) by sending us past the hand figuring into other parts of this segment of the program, about which more later.

What we've just talked about here is very complicated. We've met a lot of new variables, and we've really begun to swim in the deep water of how the poker program works. The rest of this section is more of the same, but it should become clearer as we progress.

MORE MATCHES

Now we know that we have at least a pair and that there might even be more than a pair. The next step is to figure whether we have two pairs. Although it would be easier figuring three of a kind at this point, in the long run it makes more sense the other way around. You'll see why in a minute.

FIVE-CARD DRAW

```
310    FOR N = 1 TO 4
320    FOR I = N + 1 TO 5
330    IF N = VAL ( MID$ (H$(X),2,1)) OR N
         = VAL ( MID$ (H$(X),3,1)) THEN I
         = 5
     : GOTO 350
340    IF V(X,N) = V(X,I) THEN H$(X) = "2"
         + LEFT$ (H$(X),3) + RIGHT$ ( STR$
         (N),1) + RIGHT$ ( STR$ (I),1)
     : I = 5
     : N = 4
350    NEXT I,N
360    IF V(X, VAL ( MID$ (H$(X),3,1))) = V
         (X, VAL ( MID$ (H$(X),5,1))) THEN
         H$(X) = "FOUR " + V$(X, VAL (
         MID$ (H$(X),3,1))) + "S"
     : GOTO 1360
370    IF LEFT$ (H$(X),1) = "2" AND MID$ (H
         $(X),3,1) = "1" THEN 530
380    IF LEFT$ (H$(X),1) = "2" THEN GOTO 1
         360
```

Once again we'll have only reached line 310 provided we have an H$(X) of "Pxx" or "PxxJ". Additionally, some impossible situations have already been weeded out in lines 280–300. We can work with this knowledge already in mind. It's a given in the situation that our earlier pair figuring would have given us the first pair encountered starting with the first matched card of the five. So if we held the hand:
10 CLUB
2 HEART
2 DIAMOND
3 SPADE
10 SPADE
our H$(X) would be "P15". So now you know where we stand at this moment.

Lines 310–320 take the nested loop one step farther (or better yet, one step less) than 210–220. In the earlier loops we eliminated some of the possibilities simply by going from 1 to 4 and from 2 to 5, respectively. Here we discard further impossibilities by defining I as ranging from N + 1 to 5, rather than 2 to 5 (actually, we could have done this the first time as well, but a little bit of new information goes a long way). Line 330, of course, merely eliminates duplications we don't want. And 340 goes to the heart of the matter. If this line finds another pair, then our H$(X) changes to ''2'' + ''Pxx'' + ''xx'', or ''2Pxxxx''. Using the LEFT$(3) of the original H$(X) eliminates the ''J'' at the end, if there was one, as this information is no longer important. And we now know that the first two little X's represent the numbers of the cards from 1 to 5 that are the first pair, and the second two little X's represent the numbers of the second pair. We will be using those numbers again as time develops.

Line 360 shows why we put the two-pair figuring ahead of the three of a kind: That allowed us this shortcut here. Simply enough, if the V values of both sets of pairs are equal, that means we have two pairs of the same card (two pairs of jacks, for instance): In other words, we have four of a kind. The H$(X) in this line is not an abbreviation for us but a complete spelling out that would look like, for example:
FOUR JS
Whenever a hand is finished—i.e., a pat hand where there will be no further changes—we can spell out the name of the hand in full detail, since this name will eventually be printed on the screen, and we want to save ourselves the trouble of renaming it later. As you can see, the end result of this line is a fast ticket out of the hand-figuring to the next part of this routine at 1360.

Lines 370 and 380 give us two more eliminations. The first, in 370, sets up the possibility that we might be holding a full house. The only way this is possible is if H$(X) looks like ''2P1xxx''. If the first number after the P were a 2 instead of a 1 we'd know that card 1 doesn't match anything. So here we direct programming to the full-house arena if the criterion is met. If it is not and we do not have two pairs, then we simply bow out of hand-figuring in 380 with a branch to 1360. The next line in the program is 390, and the only way we'll get there is if we're holding a pair, either ''Pxx'' or ''PxxJ''.

FIVE-CARD DRAW

```
390    FOR N = 1 TO 5
400    IF N = VAL ( MID$ (H$(X),2,1)) OR N
         = VAL ( MID$ (H$(X),3,1)) THEN 420

410    IF V(X,N) = V(X, VAL ( MID$ (H$(X),2
         ,1))) THEN H$(X) = "T" + MID$ (H$(
         X),2,2) + RIGHT$ ( STR$ (N),)
     : N = 5
420    NEXT N
430    IF LEFT$ (H$(X),1) = "T" OR RIGHT$ (
         H$(X),1) = "J" THEN 1360
450    GOTO 610
```

Here we're looking for three of a kind. We simply set up a loop and look for a V match to the pair we have already. If we get it, our H$(X) becomes

"Txxx", consisting of a "T" plus our first two little X's from the old H$(X) plus a new little X for the third card. Note that in 410, as in all our FOR . . . NEXTs, as soon as we get our condition fulfilled we jump out of the loop. In some cases this is not a logical necessity: In the present case we either have three of a kind or we don't, and all the additional figuring in the world won't change it (if we hadn't already eliminated four of a kind this would not be true). But we do it nonetheless almost by rote. If we've got what we're looking for there's no point wasting time looking for it any further.

Line 430 is the next set of exclusions. If we've got three of a kind or jacks or better, we bounce out of the figuring routine. And if we have a pair lower than jacks, the other possibility, it's off to 610 from line 450 to see if we have a four straight or four flush.

```
530    FOR N = 1 TO 5
540    FOR I = 3 TO 6
550    IF N = VAL ( MID$ (H$(X),I,1)) THEN
          I = 6
     : FLAG = 1
560    NEXT
     : IF FLAG = 1 THEN FLAG = 0
     : GOTO 590
570    IF V(X,N) = V(X, VAL ( MID$ (H$(X),3
          ,1))) THEN H$(X) = "FULL HOUSE "
          + V$(X, VAL ( MID$ (H$(X),3,1)))
          + "S"
     : N = 5
580    IF V(X,N) = V(X, VAL ( RIGHT$ (H$(X)
          ,1))) THEN H$(X) = "FULL HOUSE "
          + V$(X, VAL ( RIGHT$ (H$(X),1)))
          + "S"
     : N = 5
590    NEXT
600    GOTO 1360
```

No, we're not skipping anything here between 450 and 530. Near the end of my debugging I discovered that what once occupied this territory was a routine for figuring a full house if the H$(X) equaled "Txxx". After six months of thinking about it I finally realized that to have reached the three-of-a-kind figuring in the first place we had already eliminated any chance of having a full house. It didn't hurt having these lines here, but it certainly didn't accomplish anything. And it took me a long time even to realize it.

The 530–590 loop will be reached only if we have "2Pxxxx". The inner loop 540–560 eliminates N values equal to the little X's, setting a flag and sending us to 590, where we get our NEXT N. Lines 570 and 580 give us our full house, either matching the first pair or matching the second pair. Again you'll note the complete spelling out of the hand, e.g.:
FULL HOUSE 7S

And 1360 takes us once again out of the realm of hand-figuring, either with two pairs or a full house.

THE SNEAKY HANDS

By now we've eliminated all the matches, which was the easy part because either we had them or we didn't. From here on we get a lot more complicated for two reasons: First, the hands themselves are more complicated to analyze, and second, we have to analyze them as pieces as well as wholes. At this point of the game, four cards to a flush or an open-ended straight are quite a good hand, and we have to know if we've got them, just as we've got to know if we have the whole megillah.

To put it mildly, the figuring of a straight is one of the most convoluted operations I've ever programmed. There are a number of ways to do it, and I've used at least three in my time that I can recall offhand. The one here is no better or worse than any of them, and I apologize in advance for the confusion we're going to be going through.

FIVE-CARD DRAW

```
610    FOR N = 1 TO 5
620    FOR I = 1 TO 5
630    IF DUBCHK$(X) = "ON" AND N = S4
         THEN I = 5
     : GOTO 650
635    IF DUBCHK$(X) = "ON2" AND N = S1
         THEN I = 5
     : GOTO 650
640    IF V(X,N) = V(X,I) + 3 THEN S1 = I
     : S4 = N
     : N = 5
     : I = 5
     : FLAG = 1
650    NEXT I,N
     : IF FLAG = 1 THEN FLAG = 0
     : GOTO 760
660    IF IN(X) = 0 THEN 720
670    IN(X) = 0
680    FOR N = 1 TO 5
690    IF V(X,N) = 1 THEN V(X,N) = 14
700    NEXT
705    IF DUBCHK$(X) = "ON" THEN DUBCHK$(X)
         = "ON2"
     : GOTO 610
710    DUBCHK$(X) = ""
     : GOTO 910
720    FOR N = 1 TO 5
730    IF V(X,N) = 14 THEN V(X,N) = 1
740    NEXT
750    IN(X) = 1
     : GOTO 610
760    FOR N = 1 TO 5
```

```
770    IF V(X,N) = V(X,S1) + 2 THEN S3 = N
     : N = 5
     : FLAG = 1
780    NEXT
     : IF FLAG = 1 THEN FLAG = 0
     : GOTO 810
790    IF DUBCHK$(X) = "" THEN DUBCHK$(X)
         = "ON"
     : GOTO 610
800    GOTO 680
810    FOR N = 1 TO 5
820    IF V(X,N) = V(X,S1) + 1 THEN S2 = N
     : N = 5
     : FLAG = 1
830    NEXT
     : IF FLAG = 1 THEN FLAG = 0
     : GOTO 850
840    GOTO 680
850    H$(X) = "4S" + RIGHT$ ( STR$ (S1),)
         + RIGHT$ ( STR$ (S2),1) + RIGHT$ (
         STR$ (S3),) + RIGHT$ ( STR$ (S4),)

860    FOR N = 1 TO 5
870    IF V(X,N) = V(X,S1) + 4 THEN H$(X)
         = "STRAIGHT TO " + V$(X,N)
     : CH(X) = V(X,N)
880    IF V(X,N) = V(X,S1) - 1 THEN H$(X)
         = "STRAIGHT TO " + V$(X,S4)
     : CH(X) = V(X,S4)
890    NEXT
900    GOTO 1140
```

We'll have arrived at 610 only if we have an H$(X) of "" or a low pair. In our straight figuring we're going to start by looking for a situation where the value of one card is separated from the value of another card by a difference of 3. Since that difference could be between any two of the cards, we tediously have to figure them all in our loops at 610–620. Why do it this way? Simple. What we're looking for is a hand where two cards in the middle might give us a four straight, a 3 and a 6, say; or a 9 and a queen. If no two cards of the five are separated by an interval of 3, there is no possibility of either a full straight or a four straight, and we can get right out of this part of the figuring.

We have two other problems, however—one obvious, the other incredibly subtle. The first problem is that, in poker, aces can be either high or low, so we have to figure it both ways. At present, our aces all have a value of 14, so we'll have to change their value to 1 if they don't work out at the 14 level. That's the easy one. The second one, which will happen one time in a million, is a hand where the cards are placed precisely like this:

8 CLUB
JACK DIAMOND
6 HEART
7 CLUB
5 SPADE

Look at that one for a minute, and then look at the programming in this section. Right off the bat you can see that we do have a four straight, the 5 through the 8. No problem. But the way this programming is designed, the first thing the computer will see is the difference between the 8 of clubs and the jack of diamonds. Yes, indeed, this fits the criterion of 640, but when we go to 760 as directed, we will find no cards between the 8 and the jack, which means that there is no straight between them. Alas, unless we plan ahead, we'll miss the straight simply because of the way the cards are arranged. Needless to say, we need to do something about that.

Now, I don't want you to think I thought of this when I first wrote the program. It wasn't until I had run through a couple of hundred debugging hands that I finally saw this problem. We'll talk more about debugging this sort of program in the next chapter and how to find this sort of almost-invisible bug. For the nonce, we do know this problem exists, and line 790 is going to take care of it for us.

In the normal course of events, if the first criterion of 640 is met, the differential of 3, we'll go down to 760. Line 760 will look for another card between the two already set, now known as S4 (the higher one) and S1 (the lower one). If it finds a fit, then on to 810 to look for another piece of the straight. If not, line 790 sets a string flag for us, DUBCHK$(X), and then sends us back to the beginning of this section of programming. Line 630 is looking for that flag, and if it finds it, acts accordingly, discarding half the problem possibility that has arisen. Processing will continue, and line 705 will reset DUBCHK$(X) once again to find the other half of the problem (the fit could be off in either direction, S4+3 or S1-3). This is time-consuming processing, but it will arise only occasionally, and it must be done to ensure correct evaluation of the hands.

Meanwhile, there's the (easier?) problem of aces being high or low. We take care of that with the IN(X) flag. (IN as in INversion.) We insert that fix starting at 660, branching to 720 and resetting the V values of any aces, setting the IN(X) flag to 1, and then going back and starting over. If no straight is found, the Vs are reset, as in IN(X), starting at 670.

Where does all this pasta flow, anyhow? Good question. It flows in and out and back and forth and up and down and over and around. It works, and it wasn't easy to get it all straightened out. If you want to follow the logic in action, just program these lines alone with some dummy values (DIM the arrays first), then TRON and RUN.

Anyhow, when you get to the end of all this (line 850), you will have at least a four straight; your small pair, if any, will have been discarded. Your H$(X) will be "4Sxxxx": those little X's will tell you in order which is the lowest card up through the highest. The loop 860–890 will tell you if you have the whole straight, by determining if there's a fifth card one value lower than your first little X, or one value higher than your last little X. CH(X) in these lines is rerunning of the CHeck(X) variable used in our shuffling. For example, two players have straights. The value of CH(X) will tell us which has the higher, and therefore winning, straight. Eventually we will delineate CH(X)s for all the hands, and you'll see how we evaluate them at the end of the program.

The only way we'll have gotten this far in the program without bouncing somewhere else is if we have at least the four straight. With either a four or five straight in hand, then, we'll hit line 1140, which will figure whether we have a flush. This is a necessary step regardless of whether we have part or all of the straight. A four flush is a better hand than a four straight, and we would opt for the former if the choice is offered. And there is a very remote possibility that we might have a straight flush. This will be the time to find that out.

Having absorbed all of that, we will see that there isn't much new in the other half of the coin, the dreaded inside straight.

FIVE-CARD DRAW

```
910    REM  INSIDE STRAIGHT
920    IF X = 2 OR X = 3 OR X = 5 OR H$(X)
         < > "" THEN 1140
930    FOR N = 1 TO 5
940    FOR I = 1 TO 5
950    IF DUBCHK$(X) = "ON" AND N = S5(X)
         THEN I = 5
     : GOTO 970
955    IF DUBCHK$(X) = "ON2" AND N = S1
         THEN I = 5
     : GOTO 970
960    IF V(X,N) = V(X,I) + 4 THEN S1 = I
     : S5(X) = N
     : N = 5
     : I = 5
     : FLAG = 1
970    NEXT I,N
     : IF FLAG = 1 THEN FLAG = 0
     : GOTO 1070
980    IF IN(X) = 0 THEN 990
985    IN(X) = 0
     : GOTO 1030
990    FOR N = 1 TO 5
1000   IF V(X,N) = 14 THEN V(X,N) = 1
1010   NEXT
1020   IN(X) = 1
     : GOTO 930
1030   FOR N = 1 TO 5
1040   IF V(X,N) = 1 THEN V(X,N) = 14
1050   NEXT
1055   IF DUBCHK$(X) = "ON" THEN DUBCHK$(X)
         = "ON2"
     : GOTO 930
1060   IN(X) = 0
     : DUBCHK$(X) = ""
     : GOTO 1140
1070   FOR N = 1 TO 5
1080   FOR I = 1 TO 5
1090   IF V(X,N) = V(X,S1) + 3 AND V(X,I)
         = V(X,S1) + 2 THEN H$(X) = H$(X)
         + "IS" + RIGHT$ ( STR$ (S1),1) +
         RIGHT$ ( STR$ (I),1) + RIGHT$ (
         STR$ (N),1) + RIGHT$ ( STR$ (S5(X)
         ),1) + RIGHT$ ( STR$ (V(X,S1) + 1)
         ,1)
```

```
1100  IF V(X,N) = V(X,S1) + 3 AND V(X,I)
        = V(X,S1) + 1 THEN H$(X) = H$(X)
        + "IS" + RIGHT$ ( STR$ (S1),1) +
        RIGHT$ ( STR$ (I),1) + RIGHT$ (
        STR$ (N),1) + RIGHT$ ( STR$ (S5(X)
        ),1) + RIGHT$ ( STR$ (V(X,S1) + 2)
        ,1)
1110  IF V(X,N) = V(X,S1) + 2 AND V(X,I)
        = V(X,S1) + 1 THEN H$(X) = H$(X)
        + "IS" + RIGHT$ ( STR$ (S1),1) +
        RIGHT$ ( STR$ (I),1) + RIGHT$ (
        STR$ (N),1) + RIGHT$ ( STR$ (S5(X)
        ),1) + RIGHT$ ( STR$ (V(X,S1) + 3)
        ,1)
1120  NEXT I,N
1130  IF LEFT$ (H$(X),1) < > "I" AND DUBCH
        K$(X) = "" THEN DUBCHK$(X) = "ON"
      : GOTO 930
```

Right off the bat here you can see in line 920 that we'll be figuring this possibility only for players 1 and 4, Marvin and Betsy. Since no one else would dream of drawing to an inside straight, we needn't figure them for it. Further, since we could have arrived at this point for Marvin and Betsy with them actually holding a small pair, we also bump them with that holding by throwing in the H$(X)<>"" bit. I want you to look at this line for a minute, because there's a devil here, and it reminds me of an exam I once took in high school biology. On that exam, which was multiple-choice, there was a question something like this:

29. WHICH OF THE FOLLOWING IS FALSE?
    A. an amoeba is not free-moving
    B. a protozoa can be multicellular
    C. an amoeba is either animal or vegetable
    D. all of the above
    E. none of the above

I assure you that the groans shook the whole building when this question popped up. It is a model of disjointed logic, consisting not only of NOTs but also of NOT NOTs, NOT NOT NOTs, and even a NOT NOT OR NOT NOT. Knowing the correct answers was no clue, because if they were all correct, then the correct answer would have been E, none of the above. This is the kind of nonsense that drives students into the streets.

Anyhow, line 920 is a mild example of this disjointed logic. In Boolean terms it looks like this: IF X OR Y OR Z OR NOT A THEN B. Not so confusing, perhaps, but where the first three criteria are straightforward, the fourth is a NOT mixed in for good measure. The logic is simple enough here to work and make sense, but this kind of programming line can turn you old before your time if the conditions are complex enough. Beware of such logic, however accurate, in your programs. Sometimes it's easier to write a couple of extra lines of program than to try to figure out these monsters.

This digression notwithstanding, the inside-straight figuring mostly just recapitulates the regular straight figuring, this time looking for a gap of 4 with a missing card in between, e.g., 4 and 8, with the other cards a 5, 6, and queen—or whatever. All the IN(X) and DUBCHK$(X) business is in place as before. If we do find an inside straight, we will give it an H$(X) like this: ''ISxxxxx''. The first four little X's tell us which are the cards we're keeping, in order. The fifth X tells us the *value* of the card we'll need to draw to fill in the straight.

At the end of this segment we will slide right into the flush-figuring section, since both Marvin and Betsy would much rather have a four flush then a dreaded inside straight.

FIVE-CARD DRAW

```
1140   FOR N = 2 TO 5
1150   IF ST(X,N) < > ST(X,1) THEN N = 5
     : FLAG = 1
1160   NEXT
     : IF FLAG = 1 THEN FLAG = 0
     : GOTO 1200
1170   IF LEFT$ (H$(X),4) < > "STRA" THEN H
         $(X) = "FLUSH " + ST$(X,1) + ST$(X
         ,1) + ST$(X,1)
1180   IF LEFT$ (H$(X),4) = "STRA" THEN H$(
         X) = "STRAIGHT FLUSH"
1190   GOTO 1360
1200   FOR N = 1 TO 5
1210   FOR N2 = 1 TO 5
1220   FOR N3 = 1 TO 5
1230   IF N2 = N3 OR N = N3 OR N = N2 THEN
         1250
1240   IF ST(X,N) = ST(X,N2) AND ST(X,N) =
         ST(X,N3) THEN H$(X) = H$(X) + "3F"
         + ST$(X,N)
     : F(X) = N
     : F2 = N2
     : F3 = N3
     : N = 5
     : N2 = 5
     : N3 = 5
     : FLAG = 1
1250   NEXT N3,N2,N
     : IF FLAG = 0 THEN 1360ELSEFLAG = 0
     : GOTO 1270
1270   FOR N4 = 1 TO 5
1280   IF N4 = F2 OR N4 = F3 OR N4 = F(X)
         THEN 1300
1290   IF ST(X,N4) = ST(X,F(X)) THEN H$(X)
         = "4F" + ST$(X,F(X))
     : N4 = 5
     : FLAG = 1
1300   NEXT
     : IF FLAG = 1 THEN FLAG = 0
     : GOTO 1360
1310   IF X < > 1 AND LEFT$ (H$(X),2) = "3F
         " THEN H$(X) = ""
     : GOTO 1360
```

```
1320   IF LEFT$ (H$(X),2) = "3F" THEN 1360
1330   FOR I = 1 TO LEN (H$(X))
1340   IF MID$ (H$(X),I,2) = "3F" THEN H$(X
         ) = LEFT$ (H$(X),I - 1)
1350   NEXT
```

This is not a complicated figuring, especially compared to the straight. If we can survive the loop at 1140–1160 without branching to 1200, we definitely have a full flush. If we also have a straight, then we have a straight flush. Lines 1170–1180 take care of those possibilities.

More likely, at this point we have less than a full flush. Lines 1200–1250 give us a three-looper, looking for (shiver!) a ‘‘3F’’ H$(X). Marvin, indeed, would play a three flush if he had nothing better. Note how 1250 combines all three levels of the loop in one neat NEXT. The reason we set F(X) will become clear later, so don’t worry about it now. Keep in mind that we have to initialize new variables when we’re in the middle of a loop like this because the original variables of N, N2, and N3 will be augmented by the loop itself. If we succeed with our three flush we’ll go on to a four flush, in 1270–1300. And finally, lines 1310–1350 clean up the hands, making sure we have kept the hands of the highest value. And that’s the end of the hand-figuring section of the program.

NAMING THE HANDS

We’ve already given our hands H$(X) names, of either of two varieties. The first variety has been the shorthand name (‘‘Pxx’’ or ‘‘ISxxxxx’’) for our own internal figuring purposes. Some of those purposes we’ve already discovered; the rest we’ll see later. The second variety of names has been the fully spelled out (‘‘FOUR QS’’, ‘‘FLUSH ◇ ◇ ◇’’) for the external purpose of eventually printing them on the video screen. But we have to know more about the hands than just what they are—we also have to know how strong they are relative to each other. So we establish a hierarchy, rating the hands from 0 to 13.

FIVE-CARD DRAW

```
1360   REM    ASSIGN H(X) VALUES
1380   IF H$(X) = "" THEN H(X) = 0
1390   IF LEFT$ (H$(X),2) = "IS" THEN H(X)
         = 1
1400   IF LEFT$ (H$(X),2) = "3F" THEN H(X)
         = 2
1410   IF LEFT$ (H$(X),2) = "4S" THEN H(X)
         = 3
1420   IF LEFT$ (H$(X),2) = "4F" THEN H(X)
         = 4
1430   IF LEFT$ (H$(X),1) = "P" THEN H(X)
         = 5
     : IF LEN (H$(X)) > 3 THEN H(X) = 6
1450   IF LEFT$ (H$(X),2) = "2P" THEN H(X)
         = 7
```

```
1460  IF LEFT$ (H$(X),1) = "T" THEN H(X)
        = 8
1470  IF LEFT$ (H$(X),10) = "STRAIGHT T"
        THEN H(X) = 9
1480  IF LEFT$ (H$(X),2) = "FL" THEN H(X)
        = 10
1490  IF LEFT$ (H$(X),2) = "FU" THEN H(X)
        = 11
1500  IF LEFT$ (H$(X),2) = "FO" THEN H(X)
        = 12
1510  IF LEFT$ (H$(X),10) = "STRAIGHT F"
        THEN H(X) = 13
1520  IF REDO = 1 THEN RETURN
1540  ON X GOSUB 1570,1630,1710,1900
1550  NEXT
1560  RETURN
```

These lines do this rating job for us. A hand with nothing gets a 0, an inside straight gets a 1, a three flush a 2, and so forth. This is straightforward drudge programming, but at least it's short. The only cute thing we do is measure the LEN of any H$(X) beginning with "P"—if it's more than 3 (line 1430) we know we have at least a pair of jacks, "PxxJ", so H(X)=6. Once again in this segment we're using mirrored variables, this time H$(X) and H(X). These two are so inextricably related that it makes utmost sense to bind them like this. Of course, one is a string variable and the other a real-number variable, so as far as the computer is concerned they are worlds apart, but we know better.

Line 1520 introduces a factor we'll let ride for a minute; we'll come back to the variable REDO shortly. Line 1540 branches into separate subroutines depending on the value of X to determine each player's betting strategy for this segment of the game. Note that although X could have up to five values, there are only four GOSUBs listed. In an ON statement, if the value of X matches up with one of the possibilities (and here there are four possibilities), then it will GOSUB (or GOTO) as directed. If there is no correct possibility, in this case X=5, then the computer will simply ignore the ON . . . GOSUB statement and continue processing on the next line. We will be using this fact again later, and it is important to remember when using ONs to make sure you have enough directions for each variable, because if you don't, the computer will just keep on going (which in this case is exactly what we want). As for these particular GOSUBs, they're worth a whole section to themselves, immediately following this one. Please stay tuned.

Last, we have the NEXT statement at 1550. NEXT what? you might ask. It's NEXT X. (It's faster not to include the variables in a NEXT statement if you can avoid them. If you use them, the computer will make sure it's the right one; if you don't, it will just do the NEXT whatever comes next. The best rule is to use the variable names only when you as programmer feel you need them to keep things straight in your own head.) At this point in the program, you'll go back and do the whole thing over again for the next player. And if all the players have been taken care of, then it's goodbye, *amigo,* and back to the traffic-control section at 10000.

CREATING THE BETTING ALGORITHMS

At this point we have determined the holdings of each player, but we haven't done anything about them. It makes no difference yet whether Betsy is going to draw to an inside straight, as long as we know whether she's got that inside straight. But in a minute we're going to be betting, and that tactical information of who's going to be playing what, plus that strategic information of how much they're going to be betting on what, are going to become very important indeed. And while I was very cavalier in an earlier chapter on strategy and tactics and stated that strategy was the way one handled one's money and tactics was the way one handled one's cards, the programming does at times overlap, as we'll see now. But for the most part tactics are much more easily accommodated by the program than the strategy will be. If a given player doesn't have a hand of a certain value, he will simply drop out. 'Nuff said. Strategy is a more complicated matter.

In the real world, where nothing but human beings are sitting down at the poker table, the factors governing betting are multitudinous. We tend not only to bet our own hands but also to bet against the other players. If Marvin, who always stays in no matter what he's holding, makes a small bet, we do not worry about it. On the other hand, if Marvin, who never bluffs, makes a big bet, we tend to believe him. We respond to both who is betting and what he is betting. And in real poker we don't have too much difficulty keeping track of all the different players; the human brain has no problem monitoring four or five shifty characters, including itself. But in generating a game of computer poker we have to have a different focus for our intentions. Almost by definition our computer game is man vs. machine. We are not trying with each of our four computer-controlled players to beat each of the other computer players, which means we will be giving them no inside knowledge on each other. That is, just as the human player won't know that Betsy is a big bluffer, neither will Gerry. The difference is, while the human player will eventually learn this fact, the computer Gerry won't. Essentially each of the four computer players is playing head-on against the human, and the hell with everything else. This is a common trick in computer games. As a rule, it is almost impossible to program a single opponent as good as the average human in any game, shoot-'em-up or strategy. So what game designers do is bring in the strength of numbers against the human's strength of wit. Maybe we can't program one spaceship to outshoot the human's spaceship, but if we use 50 or 60 spaceships against the human's one, then maybe the computer will have a chance. Why do you think there are so many Space Invaders up there, and only one of you? You're just too good for any program unless that program uses brute strength.

In "Five-Card Draw" our brute strength is our four players vs. the human's one. And that brute strength is going to operate ultimately on the basis of three variables: DF(X), BF(X), and RF(X). You can think of them as Drop Factor, Bet Factor, and Raise Factor, and they will work very simply. Each one is expressed in terms of betting dollars: the Drop Factor, or DF(X), is the amount of money bet that will cause our computer-operated player to drop out—i.e., if there's enough money against him, player X will simply fold his hand. The Bet Factor, or BF(X), is the amount of money the player will open on a given hand. And the Raise Factor, the RF(X), is the amount of money the player will raise on a given hand. Let's take each player one by one.

FIVE-CARD DRAW

```
1570  REM  MARVIN
1580  IF H(1) < 8 THEN BF(1) = 1
    : RF(1) = 0
1590  IF H(1) = 8 OR H(1) = 9 THEN BF(1)
       = 2
    : RF(1) = 2
1600  IF H(1) > 9 THEN BF(1) = 3
    : RF(1) = 5
1610  DF(1) = 21
1620  RETURN
```

As you know, Marvin is going to stay in no matter what. He is a player of blunt tactics and minimal strategy. If he has a pair or two pairs (an H(1) of 7 or 8—line 1580) he will open for $1, and he won't raise. On three of a kind or a straight he'll up that to $2, with an RF(1) (Marvin is P(1)) of $2. With any pat hand he'll open for 3 and then break open his tell-all poker face by raising the maximum of $5. And he will not drop out, even if he is holding nothing (1610). Since $5 is the maximum bet and there is a three-bet limit, the maximum amount of money that can be on the table (less antes, which is how we'll be figuring it) is $20. Marvin's Drop Factor is $21. Good luck, Marvin.

FIVE-CARD DRAW

```
1630  REM  GERRY
1640  IF H(2) < 3 THEN DF(2) = 0
1650  IF H(2) = 3 OR H(2) = 4 THEN RF(2)
       = 0
    : DF(2) = 6
1660  IF H(2) > 4 AND H(2) < 8 THEN BF(2)
       = 1
    : RF(2) = 0
    : DF(2) = 4
    : IF H(2) > 5 THEN DF(2) = 14
1670  IF H(2) = 8 OR H(2) = 9 THEN BF(2)
       = 2
    : RF(2) = 1
    : DF(2) = 21
1680  IF H(2) = 10 THEN BF(2) = 3
    : RF(2) = 3
    : DF(2) = 21
1690  IF H(2) > 10 THEN BF(2) = 3
    : RF(2) = 5
    : DF(2) = 21
1700  RETURN
```

Again, Gerry is our best player. He won't even play the game through the draw if he isn't holding at least a four straight (line 1640). Even with a four straight or four flush he has a conservative DF(2) of 6 (line 1650). It's only when he is holding an opener (end line 1660) that his DF(2) is high enough to keep him in the game under any kind of serious betting. And even when he has a really good holding he doesn't tip his hand too early. He'll raise, all right, but he's still conservative; look at those RF(2)s in 1670 and 1680. He wants to keep his opponents in the game. It's only when he has a minimum hand of a full house that he starts getting nervous and raising the limit. Even Gerry isn't perfect.

FIVE-CARD DRAW

```
1710   REM  WALTER
1720   IF H(3) < 3 THEN DF(3) = 0
1730   IF H(3) = 3 OR H(3) = 4 THEN RF(3)
          = 3
     : DF(3) = 6
1740   IF H(3) < 8 AND H(3) > 4 THEN BF(3)
          = 1
     : RF(3) = 0
     : DF(3) = 4
     : IF H(3) > 5 THEN DF(3) = 10
1750   IF H(3) = 8 THEN BF(3) = 2
     : RF(3) = 2
     : DF(3) = 21
1760   IF H(3) > = 9 THEN BF(3) = 3
     : RF(3) = 3
     : DF(3) = 21
1770   IF H(3) = 6 THEN GOSUB 1800ELSE IF H
          (3) = 8 THEN GOSUB 1850
1780   RETURN
1800   FOR I = 1 TO 5
1810   IF I = VAL ( MID$ (H$(3),2,1)) OR I
          = VAL ( RIGHT$ (H$(3),1)) THEN 183
          0
1820   IF V(X, VAL ( MID$ (H$(X),2,1))) <
          > 14 AND V(3,I) = 14 THEN H$(3) =
          H$(3) + "K" + RIGHT$ ( STR$ (I),1)

     : BF(3) = 2
     : RF(3) = 4
     : DF(3) = 21
1830   NEXT
1840   RETURN
1850   FOR I = 1 TO 5
1860   IF I = VAL ( MID$ (H$(3),2,1)) OR I
          = VAL ( MID$ (H$(3),2,2)) OR I =
          VAL ( RIGHT$ (H$(3),1)) THEN 1880
1870   IF V(X, VAL ( MID$ (H$(X),2,1))) <
          > 14 AND V(3,I) = 14 THEN H$(3) =
          H$(3) + "K" + RIGHT$ ( STR$ (I),1)

     : BF(3) = 1
     : RF(3) = 4
     : DF(3) = 21
1880   NEXT
1890   RETURN
```

Because Walter is cagey, his factors are the longest to program. Like Gerry, he'll drop with a stinker hand, but look at the RF(3) in line 1730. If Walter is holding a four straight or four flush, he's going to push up that betting quite a bit—that's what betting on the come is all about. After that, he gets sneaky. Note that he doesn't push those RF(3)s on the big hands. When Walter is sitting pretty, he's going to make sure the other players are sitting in there with him.

The exception we have with Walter is his willingness to keep a kicker; hence the further subroutines at 1770–1775. If Walter is holding an ace with either a high pair or three of a kind, he's going to hold onto it in the hope of fooling his opponents. And his factors will change accordingly. The programming in these lines is similar to the earlier match figuring. Tricky guy, that Walter.

FIVE-CARD DRAW

```
1900  REM  BETSY
1910  IF H(4) < 3 THEN DF(4) = 0
1920  IF R(4) > 50 AND H(4) < 3 THEN RF(4)
        = 0
    : DF(4) = 6
1930  IF H(4) > 2 AND H(4) < 6 THEN RF(4)
        = 0
    : DF(4) = 6
1940  IF H(4) = 6 THEN BF(4) = 1
    : RF(4) = 0
    : DF(4) = 9
1950  IF H(4) = 7 OR H(4) = 8 THEN BF(4)
        = 2
    : RF(4) = 2
    : DF(4) = 21
1960  IF H(4) > 8 THEN BF(4) = 3
    : RF(4) = 3
    : DF(4) = 21
1970  RETURN
```

And finally there's Betsy, lone female at the table. She's a gut player and an odd one. For her, the size of her bankroll is just as important as the cards in her hand. Look at 1910–1920. In 1910 she is tempted to drop with a nothing hand, but in 1920, if she notices that she has more than $50 in chips (R(4)>50), she will decide to chance it after all, provided the further stakes aren't too high—that DF(4) is still pretty low. Following that, she's relatively conservative . . . but wait until you see her at the second round of betting.

BETTING THE HANDS—COMPLETING THE BETTING ALGORITHM

Now we know what everybody has in his or her hand, and we've established a pattern for betting those hands. We're ready at this point for the actual betting. And believe me, it's as complicated a segment of the program as any other, if not more so. Not because there's anything exotic about the logic—far from it—but because we have so much to keep track of at the same time. If you decide to rewrite this game without a betting limit or a limit to the number of raises, or anything along those lines, be my guest. I'll send you a get-well card at the mental institution.

FIVE-CARD DRAW

```
8000  REM  FIRST ROUND OF BETTING
8010  NB = 0
    : RD = 1
    : OP = D + 1
    : IF OP = 6 THEN OP = 1
8050  IF P(OP) = 0 THEN 8110
8060  OP = OP + 1
    : IF OP = 6 THEN OP = 1
8070  RD = RD + 1
    : IF RD = 6 THEN DROPFLAG = 1
    : D = D + 1
    : GOTO 8090
8080  GOTO 8050
8090  IF D = 6 THEN D = 1
8095  IF P(D) = 1 THEN D = D + 1
    : GOTO 8090
8100  GOSUB 29170
    : CLS
    : RETURN
```

Right at the start we have three new variables to digest here, and be forewarned that we're heading into the variable badlands. The first one is NB, as in Number of Bet. Remember, there's a three-raise limit, so if we start with an NB of 0, the maximum value it can attain is 3 on the third raise. The second variable is RD, as in RounD. A round consists of five bets—i.e., each player making one bet. At the end of any round of five bets without any raises, the betting will end, whereas any legal raise would start a new round and RD would be reset. And finally there's OP, which is going to be our counter variable so we'll know who's betting at a given time. In poker, the betting begins with the player on the dealer's left, so we start with an OP value of OP = D + 1. Since we have only five players, and player 1 happens to be the one to the left of player 5, we twist back to 1 from 6 as at the end of this line, if necessary. You'll see variations of this theme repeated endlessly in the betting sections.

The first thing we have to do in this segment is make our eliminations. If a player is out of the game, we don't want him betting. Line 8050 sends us down to the actual betting, provided player OP (P(OP)) is in the game. This may seem contrary to the mob-at-the-turn-in-the-road rule, but the betting segment runs without time problems, so it doesn't matter here. And as the game progresses and players begin to drop out, it may become the mob that is *not* in the game, rather than the other way around. Play it and see.

Following this we start to get detailed. We augment OP in line 8060, and then we augment RD in 8070. If we've gotten to this point and RD is 6, that means that no one opened, so we drop out of the subroutine altogether, set a DROPFLAG at 1, and then we augment D, the dealer, to move the deal along. Then we make sure that player D is still in the game (a P(D) = 1 would mean permanently out), and if he is no longer with us, we augment D once again. When we finally get a D still in the game we GOSUB 29170, which will clear all the variables that need clearing, and then we go back to the traffic-cop routine, where DROPFLAG is reset and we get a new shuffle and ante. These lines 8060–8100 are what I think of as bread-and-butter programming, and they're much easier to explain than to do. They require you to think of a lot of things at once—the movement of the deal, the movement of the round, the movement from 6 to 1, the elimination of nonplaying players—which are very hard to keep in mind when you're programming the first time through. So be prepared for problems in these allegedly simple areas: They can be absolute killers.

FIVE-CARD DRAW

```
8110  ON OP GOSUB 7000,7040,7080,7120,7160

8120  IF OP < > 5 THEN 8250
8130  PRINT @LOK,"What's your bet";
8140  INPUT ;B(OP)
8150  IF H(5) < 6 AND B(OP) > 0 THEN
        GOSUB 7160
    : PRINT @LOK,"You can't open!";
    : OP = 1
    : GOTO 8070
8160  IF B(OP) > 5 OR B(OP) > R(5) OR B(OP
        ) < 0 OR B(OP) < > INT (B(OP))
        THEN GOSUB 7160
    : PRINT @LOK,"Illegal bet";
    : GOSUB 63000
    : GOTO 8130
8170  IF B(OP) = 0 THEN OP = 1
    : GOTO 8070
8240  R(5) = R(5) - B(OP)
    : PT = PT + B(OP)
    : LB(OP) = B(OP)
    : GOTO 8330
8250  IF H(OP) > = 6 AND R(OP) > = BF(OP)
        THEN PRINT @LOK,B$(1) + RIGHT$ (
        STR$ (BF(OP)),1);
    : R(OP) = R(OP) - BF(OP)
    : PT = PT + BF(OP)
    : LB(OP) = BF(OP)
    : GOTO 8330
8260  IF H(OP) > = 6 THEN PRINT @LOK,B$(1)
        + "1";
    : R(OP) = R(OP) - 1
    : PT = PT + 1
    : LB(OP) = 1
    : GOTO 8330
8270  PRINT @LOK,B$(8);
    : GOSUB 63000
    : GOTO 8060
```

```
8330  BE = LB(OP)
    : ST = OP
    : RD = 1
    : GOSUB 63000
```

The next thing we do is get the first, opening bet. Line 8110 places our text at the appropriate LOK chat lines for each player, and 8120 once again disregards the-mob-at-the-turn-in-the-road rule, but again not with a noticeable increase in processing time. So let's start with the human player first, as the program does.

First we get the input of the human's bet, which we'll call B(OP), B as in Bet. Notice the format of the INPUT followed by a semicolon followed by the variable name. You'll remember that this is how we suppress carriage returns in an INPUT statement.

Lines 8150–8160 eliminate the incorrect bets. Line 8150 simply checks to make sure that if player 5 has opened, he has the cards to do it, an H(5) of greater than 5 (a 6 would mean jacks or better). If this is not the case, player 5 is admonished with a YOU CAN'T OPEN, OP is augmented (so to speak) from 5 to 1, and we go to 8070 to move things along to the next player. Line 8160 looks for other problems. In order they are: a bet greater than $5, a bet greater than the player's bankroll (R (5)), a bet less than $0, or a bet that is not an integer (e.g., $3.98). In any of these cases the decks are cleared and the player is asked once again for his bet in line 8130.

Having eliminated all the wrong possibilities, we now proceed to the correct ones. If player 5 bets $0, we go on to the next player (8170). Otherwise, his bankroll is decreased by the amount of his bet, the pot is increased by the same amount, and a new variable, LB(OP), is born, made equal to the amount of his bet. LB(OP) is going to mean the Last Bet of a given player . . . for a very simple reason. As the betting progresses and raises are made, when it comes the turn of a given player to bet, he must bet all the raises that have occurred since his last bet *minus* that last bet. Let's say player 1 bets $2. Then player 2 raises him $3. The bet to the rest of the players is $5, but when we get back to player 1, the bet is $5-LB(OP), $5 – $2 or $3. We need this information both internally, for bankroll and pot adjusting, and externally, to keep the human apprised of what's going on. The variable that would represent the *total* bet, in this example the $5, we will call BE.

Line 8250 begins handling the computer players. If the hand is an opener and the player's bankroll is larger than his BF, then he will bet his BF, with all the other variables being raised or lowered accordingly. If he's low on funds, line 8260, he'll bet just $1. Computer players will be allowed to "go light" once a game is in progress—i.e., play without enough money on hand—but if they don't win that hand, it's the end of them (the computer will automatically trace their bankrolls in negative numbers as needed). If player OP doesn't have an opener, he passes in line 8270, and we move on to the next player. Note that it is possible, just as in real life, for all five players not to have an opening hand. A little dull, perhaps, but the pot does keep growing if that is the case, which helps a little bit in making up for this lack of good cards.

The aforementioned BE is introduced in 8330, again BE being the total bet on the board. If one player bets $2 and the next player raises $3, then BE would be 5. If the next player raises 1, then BE would be 6, and so on. Although in some cases BE will translate directly to a bet as the players might see it, in others it will be keeping track of something the players have long since forgotten. It's the relationship BE − LB(OP) that's the important one. The variable STARTER—we've seen it in our info display—gives us the number of the player who has opened. It is important to know this because it is the STARTER who will open the second round of betting, not the player to the dealer's left who had been given the first opportunity to open in round 1. And after a pause, the bets continue.

FIVE-CARD DRAW

```
8340   REM  BETS CONTINUE AFTER OPENER
8350   OP = OP + 1
     : IF OP = 6 THEN OP = 1
8360   IF P(OP) < > 0 THEN RD = RD + 1
     : GOTO 8530
8370   GOSUB 31000
8380   ON OP GOSUB 7000,7040,7080,7120,7160

8390   IF OP < > 5 THEN 8480
8400   PRINT @LOK,"What's your bet?;
8410   INPUT ;B(OP)
8420   IF B(OP) > R(5) OR B(OP) > BE - LB(O
         P) + 5 OR B(OP) < 0 OR (B(OP) > BE
         - LB(OP) AND NB = 3) OR B(OP) <
         > INT (B(OP)) THEN GOSUB 7160
     : PRINT @LOK,"ILLEGAL BET";
     : GOSUB 63000
     : GOTO 8400
8430   IF B(OP) < > 0 AND B(OP) < BE - LB(O
         P) THEN GOSUB 7160
     : PRINT @LOK,"ILLEGAL BET";
     : GOSUB 63000
     : GOTO 8400
8440   IF B(OP) = 0 THEN P(5) = 10
     : GOSUB 40000
8450   R(5) = R(5) - B(OP)
     : PT = PT + B(OP)
     : RD = RD + 1
8460   IF B(OP) > BE - LB(OP) THEN RF(OP)
         = B(OP) - BE - LB(OP)
     : BE = BE + RF(OP)
     : NB = NB + 1
     : RD = 1
8470   LB(OP) = BE
     : GOTO 8520
8480   IF DF(OP) < = BE THEN PRINT @LOK,B$(
         9);
     : RD = RD + 1
     : P(OP) = 10
     : GOSUB 40000
     : GOTO 8520
```

```
8490  IF NB < 3 AND RF(OP) > 0 THEN
        PRINT @LOK,B$(3) + RIGHT$ ( STR$ (
        RF(OP)),1);
    : R(OP) = R(OP) - RF(OP) - BE + LB(OP)

    : PT = PT + RF(OP) + BE - LB(OP)
    : RD = 1
8500  IF NB < 3 AND RF(OP) > 0 THEN BE = B
        E + RF(OP)
    : LB(OP) = BE
    : NB = NB + 1
    : GOTO 8520
8510  RD = RD + 1
    : PT = PT + BE - LB(OP)
    : R(OP) = R(OP) - BE + LB(OP)
    : LB(OP) = BE
    : PRINT @LOK,B$(2);
8520  GOSUB 63000
8530  IF RD < > 5 THEN 8350
```

At the beginning of this section we simply do a little incrementing as before, and then we rerun the 31000 subroutine on the screen. We discussed 31000 in our chapter on formatting, but it might make sense now to look at it again. This is the subroutine for displaying the ongoing play information. If a STARTER exists, which it now does, the name of that player is printed. Below that is the name of the dealer, followed by a line "The bet is " BE−LB(OP), followed by the number of raises (NB, which will be 0 until a raise is encountered), followed by the amount of the pot and the amount of the human player's bankroll. This is nice information to have at all times, which is why we put it there. And we will rerun 31000 every time there's a possible change in any of these variables.

The format of this part of the betting section is similar to the actual opening but a bit more involved to cover some new possibilities. There are more possible illegal bets for the human, for instance; they take up two lines (8420–8430), and they are, in order: bet greater than bankroll, raise greater than $5, bet less than $0, a raise when there have already been three raises, bet not an integer, and (in 8430) low a bet other than $0 (which would mean a dropping out by the human player)—that is, a bet less than the correct amount he must bet. If his bet does equal $0 (line 8440), he gets a P(5) value of 10, meaning he has dropped out of this hand, and his cards are removed from the screen. And even though he is the only real player involved, the game will continue without him—a good time either to go to the bathroom or to take notes on how the other players conduct their play.

As before, if player 5 bets, the appropriate moves are made, this time in 8450. But now there's an extra wrinkle: What if player 5 decided to raise? In that case 8450 would be neither complete nor entirely accurate, so we have to handle that in 8460. If the bet is greater than BE−LB(OP), we introduce an RF for player 5, the difference between what he did bet and what he was supposed to bet. That difference is the amount of his raise. Then the BE is raised accordingly, NB is incremented, and the RD is reset at 1. But whether he raised or not, his new LB(OP) will be equal to the BE, whatever it is; hence 8470.

Now we go back to our nonhuman players. The first thing we handle is when they're going to drop out, when their DF is less than the BE in 8480. Lines 8490–8500 cover the situations where our computer player is going to raise. Provided that the three-raise limit hasn't been reached and that the player's RF is greater than 0, then these lines handle his roll, the pot, the round, the BE, the LB(OP), and the NB, respectively (in two separate lines, for programming sanity). If he's going to stay in but not raise, then 8510 does the job. Line 8530 figures whether there are any players left. If so, back to the beginning. If not . . .

FIVE-CARD DRAW

```
8540   BE = 0
     : LB(OP) = 0
     : GOSUB 31000
8550   SOFAR = 0
8560   FOR N = 1 TO 5
8570   IF P(N) < > 0 THEN SOFAR = SOFAR + 1

8580   NEXT
8590   IF SOFAR < > 4 THEN SOFAR = 0
     : RETURN
8600   FOR N = 1 TO 5
8610   IF P(N) = 0 THEN WINNER = N
8620   NEXT
8630   GOSUB 28250
     : WINFLAG = 1
     : RETURN
```

It would seem that we're ready now to go back to the traffic cop and proceed to the draw part of the game, but there's still one thing left to decide. It is not impossible that all but one player has dropped out of the game (it is impossible for all five of them to have dropped out—the DFs were designed to prevent that). So at this point we might already have a winner by default. This part of the program checks for that possibility.

We're going to use the variable SOFAR to determine this, and we'll simply loop through all the players in line 8570, incrementing SOFAR every time a player is out. Then, if SOFAR equals 4, we know we have one player left. That player's number on P(N) is made the WINNER (a variable), and processing GOSUBs to the winner's circle scenario, then WINFLAGs out of this one, and a new hand is shuffled and anted back at the traffic cop routine. We'll cover the winning scenario later in this chapter.

GETTING THE DRAW CARDS

Now that the first round of betting has transpired, the players are ready to take their draw cards. The maximum draw any player can take is three cards; although some variations of draw allow a player to take up to five cards, we are going to be much more strict in our game.

At this point we know two things about each hand, its H$(X) and its

H(X). In this part of the program you'll see why most of the H$(X)s were given the names they were and how exactly those names are used within the program (a few of the names were designed to aid in debugging, but more about that later). This is a very long part of the program, so settle in for a while.

FIVE-CARD DRAW

```
2000  REM  GET DRAW CARDS AND FIGURE HANDS

2010  DR = 26
    : X = D + 1
    : IF X = 6 THEN X = 1
2030  IF P(X) < > 0 THEN 4190
2035  FOR I = 0 TO 3
    : NC(I) = 0
    : NEXT
2040  ON X GOSUB 7000,7040,7080,7120,7160
2050  IF X = 5 THEN 3470
2060  ON H(X) GOTO 2320,2520,2910,3000,307
        0,3140,3220,3310
2070  IF H(X) > 8 THEN 3460
```

The draw section of the program goes from 2000 to 4230, but most of this breaks down to small segments, depending on the H(X) of the particular hand. The first thing we do is initialize a new variable, DR. DR (as in DRaw card) starts at 26 and is going to act as our counter variable for tracking down the cards dealt to each player. You'll remember that back in the shuffle section we created 40 cards at random; so far we have used 25 of these, five to each player, from CD$(1) to CD$(25) (and all the other C's and S's and so forth that go with each card). All we want to do here is take the next card off the pack starting with number 26; DR helps us do that successfully.

As in real poker, the first player to take his draw cards is the first player to the dealer's left still in the game. Hence 2010 sets our ubiquitous X at D+1, and as usual X is demoted to 1 if it reaches the "wrong" value of 6. Line 2030 simply sends us to a section of the program where X is augmented if P(X) is not playing in this hand. Line 2035 clears all the values of a variable we'll be looking at very shortly. Line 2040 is how we set our cursor LOK at the chat lines, and we'll discuss how we handle the human player (in line 2050) after we dispose of the nonhuman ones. Here the mob at the turn in the road was handled correctly: Of five possible x values, four pass through, and only one has to branch. The processing speed of the whole draw section varies a lot, depending on exactly what the H$(X) was to begin with, anywhere from practically instantaneous to a few seconds. Nothing to get upset about, but we don't want to make it any worse, hence our putting this segment of the program down low at 2000, and a close eye on the structure as we go along.

We've already discussed how ON works. Since H(X) can range anywhere from 0 to 13, and only eight of the possibilities are handled here, any value of H(X) less than 1 or greater than 8 will flow down to 2070 from 2060. Here values greater than 8 will branch to 3460, and a value of 0 will

continue down to the next line, which happens to be 2080. Note that we could have written our ON statement like this:
ON H(X) GOTO 2320,2520,2910,3000,3070,3140,3220,3310, 3460,3460,3460,3460,3460
The computer doesn't mind if it has to go to similar places ON different variables, but in this particular instance it's a lot easier on the programmer to do it the way I've done it rather than as above. But it is not uncommon to see:
ON I GOTO 400,500,500,600
or somesuch in a program when the situation calls for it.

FIVE-CARD DRAW

```
2080  DISC(X) = 3
    : PRINT @LOK,B$(10) + "3";
2090  FOR I = 1 TO 5
2100  FOR II = 1 TO 5
2110  IF V(X,I) < V(X,II) THEN II = 5
    : FLAG = 1
2120  NEXT
    : IF FLAG = 1 THEN FLAG = 0
    : GOTO 2140
2130  K1 = I
    : V(X,K1) = V(X,I)
    : I = 5
2140  NEXT
2150  FOR I = 1 TO 5
2155  IF I = K1 THEN 2220
2160  FOR II = 1 TO 5
2180  IF II = K1 THEN 2200
2190  IF V(X,I) < V(X,II) THEN II = 5
    : FLAG = 1
2200  NEXT
    : IF FLAG = 1 THEN FLAG = 0
    : GOTO 2220
2210  K2 = I
    : V(X,K2) = V(X,I)
    : I = 5
2220  NEXT
2250  FOR I = 1 TO 3
2255  IF I > 1 THEN NC(I) = NC(I - 1) + 1
    : GOTO 2270
2260  NC(I) = NC(I) + 1
2270  IF NC(I) = K1 OR NC(I) = K2 THEN 226
        0
2300  NEXT
2310  GOSUB 20
    : GOSUB 40
    : GOSUB 50
    : GOTO 2820
```

Our first subsegment of the draw routine is for when H(X) = 0. We will have gotten here only with Marvin (player 1) or Betsy (player 4)—anyone else would have dropped out by now. But these hardy souls, ravenous gamblers both, believe that if all else fails, simply hold onto your two highest cards and hope for the best. Marvin will do this all the time; Betsy will do it only if her bankroll is greater than $50. Aside from representing certain aspects of these players' "personalities," this mulelike stubbornness serves another purpose in the game as a whole. Having players who will stay in on anything guarantees that for much of the game our human player is assured at least one opponent. We don't want to have situations arise regularly where our human player has the only opening hand, upon learning about which all the other players immediately drop out. If our human lasts in the game long enough to beat Marvin and Betsy, he will be facing the cagier Walter and Gerry, who will drop out and let him have their dollar if this situation arises. But in the earlier stages of the game, keeping Marvin and Betsy in guarantees that the game ball will be kept rolling. We're going to make our human work for his winnings.

DISC(X) in line 2080 is yet another variable, as in DISCard, which tells us the number of cards our player will be discarding or drawing. The rest of 2080 in this case would print:
I'll take 3
the value of B$(10) + the literal string value of DISC(X). The LOK obtained at line 2040 will print this information on the appropriate chat line.

Following this we get some complicated logic. Our problem here is to find the two highest cards, and the only given we have is that no two cards are of like value (otherwise we'd have a pair and we wouldn't be here in the first place). So we set up a pair of loops looking for the highest V values among the five cards; we're going to be looking for I as the highest of the cards. In 2110 we check each card against each other card. If any II value is greater than our I value, we set a FLAG and run it again; augmenting II to 5 terminates the inner loop and the GOTO in 2120 gets the NEXT I. If no II is greater, processing will continue to 2130, where yet another new variable, K1, is initialized with the value of I, and the I loop is finished out. We now know that K1 is the highest card, or more to the point, that V(X,K1) is the highest value.

Lines 2150–2220 recapitulate the search for K1, but here we're looking for K2, the second-highest card, and the only real change is that we make sure we eliminate any chance of finding K1 twice.

So now we know which are our two highest cards, K1 and K2. But how do we use that information? We have to translate these variables into some "universal" variables we can use over and over again in any situation; these variables will be NC(0) through NC(3), which you saw earlier in 2035. In 2250–2300 we translate our knowledge of the two highest cards into a knowledge of the three lowest cards, our discards. In fact, NC(0) is used just for cowcatching; NC(1), NC(2), and NC(3) are the variables we will use to denote the numbers of the cards we will be drawing. Think of NC as New Card; it tells us exactly what we need to know when those cards are drawn. Here's where it happens:

FIVE-CARD DRAW

```
20      C$(((X * 5) - 5) + NC(1)) = C$(DR)
      : S$(((X * 5) - 5) + NC(1)) = S$(DR)
      : CD$(((X * 5) - 5) + NC(1)) = CD$(DR)
```

```
        : C(((X * 5) - 5) + NC(1)) = C(DR)
        : S(((X * 5) - 5) + NC(1)) = S(DR)
        : DR = DR + 1
        : RETURN
40        C$(((X * 5) - 5) + NC(2)) = C$(DR)
        : S$(((X * 5) - 5) + NC(2)) = S$(DR)
        : CD$(((X * 5) - 5) + NC(2)) = CD$(DR)

        : C(((X * 5) - 5) + NC(2)) = C(DR)
        : S(((X * 5) - 5) + NC(2)) = S(DR)
        : DR = DR + 1
        : RETURN
50        C$(((X * 5) - 5) + NC(3)) = C$(DR)
        : S$(((X * 5) - 5) + NC(3)) = S$(DR)
        : CD$(((X * 5) - 5) + NC(3)) = CD$(DR)

        : C(((X * 5) - 5) + NC(3)) = C(DR)
        : S(((X * 5) - 5) + NC(3)) = S(DR)
        : DR = DR + 1
        : RETURN
```

As we used to say in the sixties, wow! What an incredible trio of program statements. Of course, they are identical replicas of each other, except for the value of X in NC(X). They allow us to replace an old CD$(X) (and an old C, an old S, etc.) with a new value. Look at line 20. Let's say, for instance, that a player is going to draw one card and that that card happens to be card 26 (the value of DR). Let's further say that we are talking about player 1 and that he will be discarding his fourth card—V(1,4). In this case NC(1) would equal 4, the Number of the Card he is discarding. Fill in the blanks. C$(((1*5)−5)+4) equals 4. So C$(4) then becomes C$(DR), and in this case DR = 26, so we have now reinitialized C$(4) as C$(26). Not bad, and we won't need any new programming to know that each player, even after discards, will still hold CD$(1 through 5), or CD$(6 through 10), or whatever.

It is not inconceivable that these three lines 20–50 could have been arithmetically merged into one, but they are such marvels of detail that it seems best to keep them, and the NCs, separate. But they are placed at the beginning of the program because they are accessed numerous times, and they are very busy. Why not give them the best seat in the house?

We get to these lines from 2310, whence we go to 2820, which we'll cover later. At 2820 we go back to our original hand-figuring subroutine and do it all again, this time with our three new cards. In effect it's as if we have a new hand, and we deal with it accordingly. We'll discuss the details of that when we get to it. Meanwhile, let's proceed while NC(X) is still fresh and confused in your mind.

FIVE-CARD DRAW

```
2320      DISC(X) = 1
        : PRINT @LOK,B$(10) + "1";
2330      NC(1) = 1
2340      FOR I = 3 TO 7
```

```
2350  IF NC(1) = VAL ( MID$ (H$(X),I,1))
        THEN NC(1) = NC(1) + 1
     : I = 7
     : FLAG = 1
2360  NEXT
     : IF FLAG = 1 THEN FLAG = 0
     : GOTO 2340
2370  GOSUB 20
2380  IF C(DR - 1) = VAL ( MID$ (H$(X),7))
        THEN CH(X) = V(X,S5(X))
     : H(X) = 9
     : H$(X) = "STRAIGHT TO " + V$(X,S5(X))

     : GOTO 3750
2390  IF IN(X) = 0 THEN 2440
2400  FOR N = 1 TO 5
2410  IF V(X,N) = 1 THEN V(X,N) = 14
2420  NEXT
2430  IN(X) = 0
2440  FOR Y = 1 TO 5
2460  IF Y < > NC(1) AND C(DR - 1) = V(X,Y
        ) THEN H$(X) = "PAIR OF " + C$(DR
        - 1) + "S"
2470  NEXT Y
2480  IF LEFT$ (H$(X),1) < > "P" THEN 3750

2500  CH(X) = C(DR - 1)
     : H(X) = 6
     : IF CH(X) < 11 THEN H(X) = 5
2510  GOTO 3750
```

This section of programming concerns the possibility of an inside straight. You'll remember that this H$(X) would look like this, ISxxxxx, where the first four little x's tell us the four straight cards in order, and the last X would tell us the V value we would need in the draw card to complete the straight. Here our DISC(X) is 1, and we have to find which card from 1 to 5 we need to replace. We start with NC(1) = 1 in 2330, then loop through those first four little X's (or as they're expressed in 2350, MID$(H$(X),I,1)), making sure we don't duplicate one of them. If we do duplicate, we increment NC(1) until we've got it right. Finally, with the correct value of NC(1), we go to 20 for our draw. With our new card in hand, we check in 2380 to see if it is the right value. If so, we set our CH(X), then reset H(X) and H$(X), and we're off to 3750, where the screen is reformatted to show the actual draw (we'll describe how that looks later). If not, we reset the IN(X) flag (just in case we've been looking at a low straight) to make sure all our aces in this hand will be worth 14 rather than 1. Then we look for a pair, checking the most recent draw card against the other four in line 2460. Given the way the hands were originally figured, we have only these two possibilities when we draw to an inside straight: Either we get the straight, or we get a pair. Either way, 2480–2510 handle the movement from then on, setting the CH(X) and resetting H(X) if necessary and branching in either case to 3750, the screen formatting.

```
2520  DISC(X) = 2
    : PRINT @LOK,B$(10) + "2";
2530  FOR I = 1 TO 5
2540  IF ST$(X,I) < > ST$(X,F(X)) THEN NC(
        1) = I
    : I = 5
2550  NEXT
2560  FOR I = NC(1) + 1 TO 5
2570  IF ST$(X,I) < > ST$(X,F(X)) THEN NC(
        2) = I
    : I = 5
2580  NEXT
2590  GOSUB 20
    : GOSUB 40
2600  IF S$(DR - 2) < > ST$(X,F(X)) OR S$(
        DR - 1) < > ST$(X,F(X)) THEN 2820
2620  H$(X) = "FLUSH " + S$(DR - 1) + S$(D
        R - 1) + S$(DR - 1)
2630  REM   CHECK FOR STRAIGHT FLUSH
2635  IN(X) = 0
2640  FOR Z1 = 1 TO 5
2641  FOR Z5 = 1 TO 5
2642  IF V(X,Z5) = V(X,Z1) + 4 THEN S1 = Z
        1
    : S5 = Z5
    : Z1 = 5
    : Z5 = 5
    : FLAG = 1
2643  NEXT Z5,Z1
    : IF FLAG = 1 THEN FLAG = 0
    : GOTO 2649
2644  IF IN(X) = 1 GOSUB 2740
    : GOTO 3750
2645  FOR I = 1 TO 5
2646  IF V(X,I) = 14 THEN V(X,I) = 1
2647  NEXT
2648  GOTO 2640
2649  FOR Z2 = 1 TO 5
2650  IF V(X,Z2) = V(X,S1) + 1 THEN S2 = Z
        2
    : Z2 = 5
    : FLAG = 1
2651  NEXT
    : IF FLAG = 1 THEN FLAG = 0
    : GOTO 2653
2652  GOSUB 2740
    : GOTO 3750
2653  FOR Z3 = 1 TO 5
2654  IF V(X,Z3) = V(X,S2) + 1 THEN S3 = Z
        3
    : Z3 = 5
    : FLAG = 1
2655  NEXT
    : IF FLAG = 1 THEN FLAG = 0
    : GOTO 2657
```

```
2656  GOSUB 2740
    : GOTO 3750
2657  FOR Z4 = 1 TO 5
2658  IF V(X,Z4) = V(X,S3) + 1 THEN H$(X)
        = "STRAIGHT FLUSH"
    : Z4 = 5
    : FLAG = 1
2659  NEXT
    : IF FLAG = 0 THEN GOSUB 2740ELSEFLAG
        = 0
    : H(X) = 13
2730  GOTO 3750
2740  IN(X) = 0
2741  FOR I = 1 TO 5
2742  IF V(X,I) = 1 THEN V(X,I) = 14
2743  NEXT
2745  FOR I = 1 TO 5
2750  FOR N = 1 TO 5
2760  IF V(X,I) < V(X,N) THEN N = 5
    : FLAG = 1
2770  NEXT
    : IF FLAG = 1 THEN FLAG = 0
    : GOTO 2790
2780  CH(X) = V(X,I)
    : I = 5
2790  NEXT
    : IF CH(X) = 0 THEN CH(X) = V(X,5)
2800  H(X) = 10
    : RETURN
```

The programming above handles our three-flush situation. Two cards are discarded, and in 2540 and 2570 we get the two cards that are not of the correct flush suit (ST$(X,F(X))), which we set back in the hand-figuring routine. If either of these new draw cards is not in the correct suit, we branch to 2820, whence the whole hand will be refigured with the new cards. The reason for this is that anything from a straight to three of a kind could possibly have occurred with that draw of two cards, and we want to cover all the possibilities, unlike the nice, neat leftover pair possibility after we had eliminated our inside straight earlier.

If we do have a flush, we so note it in 2620, and then we have a lot more business to take care of. First we have to see if in fact we have a straight flush, which is not at all impossible considering the draw of two cards. Lines 2630–2659 handle this possibility for us. If at any time we fall out of the possibility, we branch to 2740, where we reset any aces if necessary, then get our CH(X) and H(X) values. And then we go to 3750 for screen formatting. We don't pull any CH(X) if we do have the straight flush. The odds of this happening are remote but possible, so we're ready for that; the odds of *two* people getting a straight flush in one hand are so insignificant that they aren't even worth bothering about.

FIVE-CARD DRAW

```
2820  REDO = 1
    : GOSUB 100
    : REDO = 0
```

```
2840  IF H(X) = 6 THEN CH(X) = V(X, VAL (
        MID$ (H$(X),2,1)))
    : H$(X) = "PAIR OF " + V$(X, VAL (
        MID$ (H$(X),2,1))) + "S"
2850  IF H(X) = 7 AND VAL ( MID$ (H$(X),3,
        1)) > VAL ( MID$ (H$(X),5,1))
        THEN CH(X) = V(X, VAL ( MID$ (H$(X
        ),3,1)))
    : H$(X) = "TWO PAIRS "
2860  IF H(X) = 7 AND VAL ( MID$ (H$(X),3,
        1)) < VAL ( MID$ (H$(X),5,1))
        THEN CH(X) = V(X, VAL ( MID$ (H$(X
        ),5,1)))
    : H$(X) = "TWO PAIRS"
2870  IF H(X) = 8 THEN CH(X) = V(X, VAL (
        RIGHT$ (H$(X),1)))
    : H$(X) = "THREE " + V$(X, VAL (
        RIGHT$ (H$(X),1))) + "S"
2880  IF H(X) = 9 THEN CH(X) = V(X,F3)
2890  IF H(X) = 11 OR H(X) = 12 THEN
        GOSUB 32000
2900  GOTO 3750
```

This is the part of the program from which we branch back to the original hand-figuring. First we set a new variable flag, REDO, at 1, then we GOSUB 100 (not 90, the actual beginning of the subroutine, because that line begins the FOR X = 1 TO 5 loop, and here we're interested only in one particular already-defined X). You'll remember encountering REDO in that earlier part of the program. By setting this flag we ensure that we will return to this part of the program without having to go through unnecessary parts of the earlier subroutine. What we've done by using this flag this way is turned one subroutine into two subroutines, using the same lines of programming for the same ends but from different starting and finishing points. (That multiple usage is what subroutines are all about.)

Lines 2840–2890 handle the H$(X) names that wouldn't have been derived from the hand-figuring begun at 100, as well as putting their CH(X) values on them. The 32000 GOSUB is more of the same and looks like this:

FIVE-CARD DRAW

```
32000 REM   FULL HOUSE AND FOUR OF A KIND
        CH(X) ROUTINE
32010 CH$ = RIGHT$ (H$(X),2)
32020 IF CH$ = "AS" THEN CH(X) = 14
    : GOTO 32140
32030 IF CH$ = "KS" THEN CH(X) = 13
    : GOTO 32140
32040 IF CH$ = "QS" THEN CH(X) = 12
    : GOTO 32140
32050 IF CH$ = "JS" THEN CH(X) = 11
    : GOTO 32140
32060 IF CH$ = "TS" THEN CH(X) = 10
    : GOTO 32140
32070 CH(X) = VAL ( MID$ (H$(X), LEN (H$(X
        )) - 1,1))
32140 RETURN
```

Here we read the ends of the H$(X) names (As, Ks, etc.) to derive the CH(X). The CH$(X) is merely a one-time use variable for this operation.

FIVE-CARD DRAW

```
2910   DISC(X) = 1
     : PRINT @LOK,B$(10) + "1";
2920   NC(1) = 1
2930   FOR I = 3 TO 7
2940   IF NC(1) = VAL ( MID$ (H$(X),I,1))
         THEN NC(1) = NC(1) + 1
     : I = 7
     : FLAG = 1
2950   NEXT
     : IF FLAG = 1 THEN FLAG = 0
     : GOTO 2930
2960   GOSUB 20
2970   IF C(DR - 1) = V(X, VAL ( MID$ (H$(X
         ),3,1))) - 1 THEN CH(X) = V(X,
         VAL ( MID$ (H$(X),6,1)))
     : H$(X) = "STRAIGHT TO " + V$(X, VAL (
         MID$ (H$(X),6,1)))
     : H(X) = 9
     : GOTO 3750
2980   IF C(DR - 1) = V(X, VAL ( MID$ (H$(X
         ),6,1))) + 1 THEN H$(X) = "STRAIGH
         T TO " + C$(DR - 1)
     : CH(X) = C(DR - 1)
     : H(X) = 9
     : GOTO 3750
2985   IF C(DR - 1) = 14 AND VAL ( MID$ (H$
         (X),3,1)) = 2 THEN C(DR - 1) = 1
     : IN(X) = 1
     : GOTO 2970
2990   V(X,NC(1)) = C(DR - 1)
     : GOTO 2390
```

Our next possible hand is the four straight, with an H$(X) of ‘‘4Sxxxx’’, the little X’s the numbers of the cards in the player’s hand, in order, that contain the straight values. After getting our NC(I) as before through augmentation in 2940, we plug in the C value of DR−1, our draw card, in lines 2970–2980. We do it top and bottom, since the straight could be hit either way. Line 2985 looks for a low ace for a 1-to-5 straight. If we get our straight, fine, if not, back to 2390, where IN(X) is reset and we look for the pair, our only possibility at this point (there’s no way of having a flush; that possibility was eliminated in the original hand-figuring where a four flush would have taken precedence over a four straight).

FIVE-CARD DRAW

```
3000   DISC(X) = 1
     : PRINT @LOK,B$(10) + "1";
3010   FOR I = 1 TO 5
```

```
3020  IF ST$(X,I) < > ST$(X,F(X)) THEN NC(
        1) = I
     : I = 5
3030  NEXT
3040  GOSUB 20
3050  IF S$(DR - 1) = ST$(X,F(X)) THEN 262
        0
3060  GOTO 2820
```

Here we're looking for the match to our four flush. If we get it, then on to 2620 to check for the straight flush, etc. If not, it's up to 2820 and a REDO. We could have a pair, and we could even have a straight. A full REDO will handle both for us.

FIVE-CARD DRAW

```
3070  DISC(X) = 3
     : PRINT @LOK,B$(10) + "3";
3080  FOR I = 1 TO 3
3090  NC(I) = NC(I - 1) + 1
3100  IF NC(I) = VAL ( MID$ (H$(X),2,1))
        OR NC(I) = VAL ( MID$ (H$(X),3,1))
        THEN NC(I) = NC(I) + 1
     : GOTO 3100
3110  NEXT
3120  GOSUB 20
     : GOSUB 40
     : GOSUB 50
     : GOTO 2820
```

Lines 3070–3120 are for when we have an H(X) of 5, a small pair. We find our NCs in the usual way and once again do a whole REDO. The next segment handles jacks or better:

FIVE-CARD DRAW

```
3140  IF LEN (H$(X)) = 4 THEN 3070
3150  DISC(X) = 2
     : PRINT @LOK,B$(10) + "2";
3160  FOR I = 1 TO 2
3170  NC(I) = NC(I - 1) + 1
3180  IF NC(I) = VAL ( MID$ (H$(X),2,1))
        OR NC(I) = VAL ( MID$ (H$(X),3,1))
        OR NC(I) = VAL ( RIGHT$ (H$(X),1))
        THEN NC(I) = NC(I) + 1
     : GOTO 3180
3190  NEXT
3200  GOSUB 20
     : GOSUB 40
     : GOTO 2820
```

You'll remember here that our H$(X) in this situation will look like ''PxxJ''—with one exception. If Walter is keeping a kicker it will look like ''PxxJKx''—with a LEN(H$(X)) of 6. Hence line 3140. If we have a regular pair of jacks or better, it can be handled just as it was earlier. But if Walter is keeping the kicker, then his DISC(X) is 2 and we proceed accordingly. But once again we go the REDO route, in any case.

FIVE-CARD DRAW

```
3220  PRINT @LOK,B$(10) + "1";
    : DISC(X) = 1
    : NC(1) = 1
3240  FOR I = 3 TO 7
3250  IF NC(1) = VAL ( MID$ (H$(X),I,1))
        THEN NC(1) = NC(1) + 1
    : I = 7
    : FLAG = 1
3260  NEXT
    : IF FLAG = 1 THEN FLAG = 0
    : GOTO 3240
3270  GOSUB 20
3280  IF C(DR - 1) = V(X, VAL ( MID$ (H$(X
        ),3,1))) THEN H$(X) = "FULL HOUSE,
         " + C$(DR - 1) + "S"
    : CH(X) = C(DR - 1)
    : H(X) = 11
    : GOTO 3750
3290  IF C(DR - 1) = V(X, VAL ( MID$ (H$(X
        ),5,1))) THEN H$(X) = "FULL HOUSE,
         " + C$(DR - 1) + "S"
    : CH(X) = C(DR - 1)
    : H(X) = 11
    : GOTO 3750
3300  GOTO 2850
```

As you can see, we're speeding up our discussion now, since much of this programming is repetitious. In fact, much of *all* programming is repetitious, and it's the ability to overcome the tedium and keep going that separates the hackers from the hacks. In a way, what we're doing now is much like the conversation with the computer segment in "Space Derelict!," the same thing over and over again, only with different values plugged in. A fact of life, I'm afraid, but bear with it a little while longer. There are still a few tricks up our sleeves that are yet to be unveiled.

Lines 3220–3300 handle our draw to two pairs. We don't have to REDO here because either our draw matches one of our pairs for a full house or it doesn't. If we have only the pairs, we go to 2850 in 3300 to get our H$(X) set up.

FIVE-CARD DRAW

```
3310  IF LEN (H$(X)) > 4 THEN PRINT @LOK,B
        $(10) + "1";
    : DISC(X) = 1
3320  IF LEN (H$(X)) = 4 THEN PRINT @LOK,B
        $(10) + "2";
    : DISC(X) = 2
```

```
3340  FOR I = 1 TO 2
3350  NC(I) = NC(I - 1) + 1
3360  IF NC(I) = VAL ( MID$ (H$(X),2,1))
        OR NC(I) = VAL ( MID$ (H$(X),3,1))
        OR NC(I) = VAL ( MID$ (H$(X),4,1))
        THEN NC(I) = NC(I) + 1
    : GOTO 3360
3370  NEXT
3380  GOSUB 20
    : IF LEN (H$(X)) > 4 THEN 3430ELSE
        GOSUB 40
3400  IF C(DR - 1) = C(DR - 2) THEN H$(X)
        = "FULL HOUSE, " + V$(X, VAL (
        RIGHT$ (H$(X),1))) + "S"
    : H(X) = 11
    : GOSUB 32000
3410  IF C(DR - 1) = V(X, VAL ( RIGHT$ (H$
        (X),1))) OR C(DR - 2) = V(X, VAL (
        RIGHT$ (H$(X),1))) THEN H$(X) = "F
        OUR " + V$(X, VAL ( RIGHT$ (H$(X),
        1))) + "S"
    : GOSUB 32000
3420  IF LEFT$ (H$(X),1) = "T" THEN CH(X)
        = V(X, VAL ( MID$ (H$(X),2,1)))
    : H$(X) = "THREE " + V$(X, VAL ( MID$
        (H$(X),2,1))) + "S"
3425  GOTO 3750
3430  IF C(DR - 1) = 14 THEN H$(X) = "FULL
         HOUSE, " + V$(X, VAL ( MID$ (H$(X
        ),2,1))) + "S"
    : GOSUB 32000
3440  IF C(DR - 1) = V(X, VAL ( MID$ (H$(X
        ),2,1))) THEN H$(X) = "FOUR " + V$
        (X, VAL ( MID$ (H$(X),2,1)))
    : GOSUB 32000
3450  GOTO 3420
```

Remember Walter and his kicker? Well, he might have done it again with three of a kind; hence lines 3310–3320. All our H$(X) naming is handled in this segment, with the appropriate branching for Walter and the not Walters. And finally:

FIVE-CARD DRAW

```
3460  PRINT @LOK,"I'LL PLAY THESE.";
    : GOSUB 63000
    : GOSUB 27180
    : GOTO 4190
```

If our computer player has a pat hand and does not need to draw, we get his CH(X) at 27180, and we go to 4190. We don't have to format the screen for this one; if computers could grin, this is when they'd be doing it. And thus ends our search for draw cards for the computer-generated players.

DRAWING THE CARDS ON THE SCREEN

This is a very short routine, and we'll skip down to it now before handling the human player's draw, since this flows more logically from what we've just been talking about. The problem we have here is that each player has said how many cards he wants, and that is what our human player has seen on the screen in the appropriate chat lines. We've also done the draw interally and refigured the hands, which is neither here nor there to our human player. Now we have to put in a little drama . . . we have to make it look as though our players are actually drawing cards. So we're going to erase the appropriate number of card backs, those empty rectangles, and redraw them on the screen. And now that we have screen formatting down to a science, we can do it most efficiently.

FIVE-CARD DRAW

```
3750  LOK = 412 + ((X - 1) * 20)
    : IF X = 4 THEN LOK = 1372
3760  FOR I = 1 TO DISC(X)
    : PRINT @LOK,ERAS$;
    : LOK = LOK - 83
    : NEXT
3770  PRINT @LOK,ART$;
    : LOK = LOK + 83
    : GOSUB 63020
3780  FOR I = 1 TO DISC(X)
    : PRINT @LOK,ART$;
    : LOK = LOK + 83
    : GOSUB 63020
    : NEXT
```

The logic here is similar to our earlier erasing and redrawing, with one difference, line 3770. Here we're redrawing ART$ over the position of the last card still held by the player. Since each ART$ partly overlays the preceding one, the points at which the overlaps occur will have been replaced by one of the blanks in ERAS$, which doesn't look very good on the screen. This quick doctoring is completely undetectable to the human player. In sum, we haven't really done anything in this segment, but it looks as though we have, and sometimes that's just as important.

THE HUMAN GETS HIS CARDS

Needless to say, we handle the human draw quite differently from the computer ones. Here the choice of what to do is entirely up to the player, and all we have to do is provide clear directions to the player. Our problem is that, given the tightness of the screen format, there's nowhere for these directions to go, so we have to engineer things just right so the player doesn't curse us for ruining his game.

```
3470   PRINT @LOK,"How many cards";
     : INPUT ;DISC(5)
3480   GOSUB 7160
3490   IF DISC(5) < > INT (DISC(5)) OR DISC
          (5) > 3 OR DISC(5) < 0 THEN GOTO 3
          470
3500   IF DISC(5) = 0 THEN PRINT @LOK,"Okay
          .";
     : GOTO 4190
3510   I = 0
3520   FOR N = 21 TO 25
3525   GOSUB 7160
3530   PRINT @LOK,"Replace " + C$(N) + S$(N
          ) + " (Y/N)";
3540   INPUT ;AN$
3550   IF AN$ = "N" THEN 3590
3560   IF AN$ < > "Y" THEN 3525ELSECD$(N)
          = CD$(DR)
     : C$(N) = C$(DR)
     : S$(N) = S$(DR)
     : C(N) = C(DR)
     : S(N) = S(DR)
3570   DR = DR + 1
3580   I = I + 1
     : IF I = DISC(5) THEN N = 25
3590   NEXT
3600   IF I < > DISC(5) THEN PRINT @LOK,"Yo
          u only took ";I;". Let's try again
          .";
     : GOSUB 63000
     : PRINT @LOK,B$(12) + B$(12);
     : GOTO 3510
3610   LOK = 1060
3620   FOR N = 21 TO 25
3630   PRINT @LOK,CD$(N);
3640   GOSUB 63020
3650   LOK = LOK + 83
3660   NEXT
3670   GOTO 4190
```

The first thing we do is get the input of DISC(5) from the player in line 3470. This will, of course, appear on the human's chat line. Line 3490 cowcatches bad inputs and sends us back to 3470, while 3500 simply takes us out of here if the human stands pat. But if the human does indicate a desire to draw cards, we are going to go down the line of his cards and display each of them one at a time, asking him if he wants to replace this particular card. We do this by setting the loop in 3520 to display the concatenated C$(N) + S$(N) values for each card. If he says no, we proceed to the NEXT N. If he says yes, we reset all his C, S, etc., values as we did with the computer players.

We have to be very sure that the program allows only the correct number of inputs, matching the number of DISC(5). That is where the I variable comes in (lines 3510 and 3580). First we set I to 0, then we increment it every time the human takes a new card. When I equals DISC(5) we exit the loop. In this way we don't ask for more cards than we need.

But what happens if the loop ends and I does not equal DISC(5), i.e., the player hasn't chosen enough cards? This contingency is handled in line 3600. Now we can be sure that the player will get just the number of cards he is entitled to. The actual printing of the cards, lines 3620–3660, is the same as the original printing of the human's CD$(X)s.

Last but not least in the draw segment comes this:

FIVE-CARD DRAW

```
4190  X = X + 1
4200  IF X = 6 THEN X = 1
4210  IF X = 1 AND D < > 5 THEN 2030
4220  IF X < > 1 AND X < > D + 1 THEN 2030

4230  RETURN
```

This merely augments our X values appropriately to keep the hand moving from one player to the next. Lines 4210–4220 keep us going if there are more players to come, and 4230 RETURNs us if the draw segment is over.

SETTING THE NEW PATTERNS OF BETTING

Our next subroutine, as indicated by our line 10000 traffic director, begins at line 5000. Here we will update the various BF(X), DF(X), and RF(X) variables based on the new hands held by the computer players. This section is identical in structure to the earlier version back at line 1570, but the values of the variables will be radically different from the first time around. The most dramatic difference is the addition of bluffing elements.

FIVE-CARD DRAW

```
5000  FOR X = 1 TO 4
5010  IF P(X) < > 0 THEN 5030
5020  ON X GOSUB 5050,5110,5190,5270
5030  NEXT
5040  RETURN
```

The introduction to this section is written in classic Basic format. We set our loop in 5000, pull out our exceptions in 5010, do an ON . . . GOSUB in 5020, a NEXT in 5030, and a RETURN in 5040. Nothing could be simpler to understand, and nothing could so easily show the concept of subroutine in its best light. Here we have boiled down our processing to its essential ingredients: We don't fool around with REMs or complicated, convoluted logic, we don't do any processing of redundant or unimportant data. We do just what we're supposed to do and then RETURN. Would that all logic and programming were so simple.

FIVE-CARD DRAW

```
5050   REM  MARVIN
5060   IF H(1) < 6 THEN DF(1) = 0
     : BF(1) = 0
5070   IF H(1) < 8 AND H(1) > 5 THEN BF(1)
         = 1
     : RF(1) = 0
     : DF(1) = 14
5080   IF H(1) > 7 AND H(1) < 10 THEN BF(1)
         = 2
     : RF(1) = 2
     : DF(1) = 21
5090   IF H(1) > 9 THEN BF(1) = 3
     : RF(1) = 5
     : DF(1) = 21
5100   RETURN
```

Good old Marvin hasn't changed much now that he's taken his draw. Of course, now he doesn't stay in on anything—if H(1) is less than jacks or better, he'll drop out (5060)—but the same uncomplicated, blunt style of betting is still readily apparent. Note that here and following we must be sure to set BF(X) as 0 (line 5060) if our player has a bum hand. Otherwise we might have a lingering BF(X) from the first round of betting, which would keep our player in the game when he's not supposed to be there.

FIVE-CARD DRAW

```
5110   REM  GERRY
5120   IF H(2) = 3 OR H(2) = 4 THEN BL(X)
         = RND (10)
     : IF BL(X) < 3 THEN BF(2) = 2
     : RF(2) = 3
     : DF(2) = 14
     : RETURN
5130   IF H(2) < 6 THEN DF(2) = 0
     : BF(2) = 0
5140   IF H(2) = 6 OR H(2) = 7 THEN BF(2)
         = 1
     : RF(2) = 0
     : DF(2) = 8
5150   IF H(2) = 8 OR H(2) = 9 THEN BF(2)
         = 2
     : RF(2) = 1
     : DF(2) = 12
     : IF H(2) = 9 THEN DF(2) = 21
5160   IF H(2) = 10 THEN BF(2) = 3
     : RF(2) = 3
     : DF(2) = 21
5170   IF H(2) > 10 THEN BF(2) = 3
     : RF(2) = 5
     : DF(2) = 21
5180   RETURN
```

Gerry, on the other hand, now introduces the possibility of bluffing with a bum hand of H(2) = 3 or H(2) = 4 (incompleted four straight and four flush, respectively). We do this in 5120 by rolling a random number for the new variable BL(X), as in BLuff. We arbitrarily give Gerry two chances out of 10 to run a bluff and then assign betting variables to him if he hits one of those two chances. Gerry's bluffs are going to be rare, but they still will be a distinct possibility—which is the way any good player approaches draw poker. And he will be conservative about them, playing up to only $14 on his DF. He's not out to cut his own throat in these last-ditch bluff attempts—he just wants you, the human, to drop out.

Without the bluffing, Gerry's betting proceeds much as it did before, a tad cautious but nonetheless reasonable. In line 5150 you see in the last statement how we manage to give two different values to DF(2) for the two different H(2)s without having to break out of the programming line. Remember, the more we can squeeze behind one line number, the better.

FIVE-CARD DRAW

```
5190  REM  WALTER
5200  IF H(3) < 3 THEN DF(3) = 0
    : BF(3) = 0
5210  IF H(3) = 3 OR H(3) = 4 THEN BL(X)
        = RND (10)
    : IF BL(X) = 7 THEN BF(3) = 2
    : RF(3) = 3
    : DF(3) = 10
5220  IF BL(X) < > 7 AND H(3) < 6 THEN DF(
        3) = 0
    : BF(3) = 0
5230  IF H(3) < 8 AND H(3) > 5 THEN BF(3)
        = 1
    : RF(3) = 0
    : DF(3) = 7
5240  IF H(3) = 8 THEN BF(3) = 2
    : RF(3) = 2
    : DF(3) = 21
5250  IF H(3) > = 9 THEN BF(3) = 3
    : RF(3) = 3
    : DF(3) = 21
5260  RETURN
```

Walter too will bluff, but he does it all wrong. He does it once in a blue moon (line 5210), and then, given his DF(3) of 10, he hardly sticks with it. It's as if he makes his bet as though he had filled the straight or flush, then thinks better of it and drops out. If our human player can spot this trait it will make Walter that much more "human."

FIVE-CARD DRAW*

```
5270  REM  BETSY
5280  IF H(4) < 9 THEN BL(X) = RND (4)
    : IF BL(X) = 4 THEN BF(4) = 3
    : RF(4) = 3
    : DF(4) = 21
    : GOTO 5340
```

```
5290  IF H(4) < 6 THEN DF(4) = 0
    : BF(4) = 0
5300  IF H(4) = 6 THEN BF(4) = 2
    : RF(4) = 0
    : DF(4) = 8
5310  IF H(4) = 7 THEN BF(4) = 2
    : RF(4) = 1
    : DF(4) = 10
5320  IF H(4) = 8 THEN BF(4) = 2
    : RF(4) = 2
    : DF(4) = 12
5330  IF H(4) > 8 THEN BF(4) = 3
    : RF(4) = 3
    : DF(4) = 21
5340  IF R(4) > 50 AND DF(4) > 0 THEN BF(4
        ) = BF(4) + 1
    : RF(4) = RF(4) + 1
    : IF R(4) > 100 THEN BF(4) = BF(4) + 1

    : RF(4) = RF(4) + 1
    : DF(4) = DF(4) + 1
5350  RETURN
```

On the other hand, Betsy bluffs like a house afire—one time out of four! And she bluffs with any hand from H(4)=0 to H(4)=8 (two pairs). Very unpredictable and even a little self-destructive, but you've got to give her credit for guts. To complicate things further, she also keeps a constant eye on her stack of chips. Take a look at 5340. If her bankroll is greater than $50, she'll bet even more heavily. And if it's greater than $100, more heavily still (note that all her RFs and BFs are designed so she won't bet past the limit). A difficult opponent to read at any time, and for the human, in some respects as hard to beat as Gerry, provided the cards go right for her. If they go wrong, she'll burn herself out. Again, this fluidity makes her that much more human in the eyes of the real human player.

THE SECOND ROUND OF BETTING

This subroutine is designed much the same as the first round of betting. Unfortunately, we cannot use that first subroutine a second time because originally we had the very different factor of opening the bidding, which we don't have here. But nonetheless some of these lines are directly copied.

FIVE-CARD DRAW

```
9000  REM  SECOND ROUND OF BETTING
9020  FOR OP = 1 TO 5
    : LB(OP) = 0
    : NEXT
9050  OP = STARTER
    : RD = 1
    : NB = 0
9060  GOTO 9080
```

```
9070  OP = OP + 1
    : IF OP = 6 THEN OP = 1
9080  IF P(OP) < > 0 THEN RD = RD + 1
    : GOTO 9260
9090  GOSUB 31000
9100  ON OP GOSUB 7000,7040,7080,7120,7160

9110  IF OP < > 5 THEN 9200
```

Once again we clear a few variables and initialize a few others. OP=STARTER is in 9050 because the second round of betting begins with the original opener, not the player to the left of the dealer. And we move the play along through each of the five players, incrementing the RD round as necessary. Line 9250 (from 9080) is the continuation of the incrementation process.

FIVE-CARD DRAW

```
9120  PRINT @LOK,"What's your bet";
9130  INPUT ;B(OP)
9140  IF B(OP) > R(5) OR B(OP) > BE - LB(O
        P) + 5 OR B(OP) < 0 OR (B(OP) > BE
        - LB(OP) AND NB = 3) OR B(OP) <
        > INT (B(OP)) THEN GOSUB 7160
    : PRINT @LOK,"ILLEGAL BET";
    : GOSUB 63000
    : GOTO 9120
9150  IF B(OP) < > 0 AND B(OP) < BE - LB(O
        P) THEN GOSUB 7160
    : PRINT @LOK,"ILLEGAL BET";
    : GOSUB 63000
    : GOTO 9120
9160  IF B(OP) = 0 AND BE > 0 THEN P(5) =
        10
    : GOSUB 40000
9170  R(5) = R(5) - B(OP)
    : PT = PT + B(OP)
    : RD = RD + 1
    : IF BE = 0 THEN BE = B(OP)
    : RD = 1
9180  IF B(OP) > BE - LB(OP) THEN RF(OP)
        = B(OP) - BE - LB(OP)
    : BE = BE + RF(OP)
    : IF BE < > 0 THEN NB = NB + 1
    : RD = 1
9190  LB(OP) = BE
    : GOTO 9250
```

Here our human bets his hand. In 9140 and 9150 he's up against all the restrictions as before. Note that if his LB(OP) = 0 it doesn't affect these lines; in fact, they are playable whether or not he or any other player has already bet in this round. That was to save ourselves some of the longer programming we had in the first round of betting. This time we can work it that much cleaner because we don't have that opening factor.

```
9200   IF BF(OP) > 0 AND NB = 0 AND BE = 0
         THEN PRINT @LOK,B$(4) + RIGHT$ (
         STR$ (BF(OP)),1);
     : LB(OP) = BF(OP)
     : BE = BF(OP)
     : RD = 1
     : PT = PT + BE
     : R(OP) = R(OP) - BE
     : GOTO 9250
9210   IF DF(OP) < BE THEN PRINT @LOK,B$(9)
         ;
     : RD = RD + 1
     : P(OP) = 10
     : GOSUB 40000
     : GOTO 9250
9220   IF NB < 3 AND RF(OP) > 0 THEN
         PRINT @LOK,B$(3) + RIGHT$ ( STR$ (
         RF(OP)),1);
     : R(OP) = R(OP) - RF(OP) - BE + LB(OP)

     : RD = 1
9230   IF NB < 3 AND RF(OP) > 0 THEN PT = P
         T + RF(OP) + BE - LB(OP)
     : BE = BE + RF(OP)
     : LB(OP) = BE
     : NB = NB + 1
     : GOTO 9250
9240   LB(OP) = BE - LB(OP)
     : RD = RD + 1
     : PT = PT + LB(OP)
     : R(OP) = R(OP) - LB(OP)
     : PRINT @LOK,B$(2);
```

The same holds true for our computer players. The programming here is very tight; line 9200, for instance, seems to have about 9200 variables in it. But they're all there where they're supposed to be, raising this value, lowering that one, initializing the other. Since we've already explained what's going on here, we won't go into detail now.

FIVE-CARD DRAW

```
9250   GOSUB 63000
9260   IF RD < > 5 THEN 9070
9270   BE = 0
     : LB(OP) = 0
     : GOSUB 31000
9280   REM  CHECK FOR DEFAULT WINNER
9300   FOR N = 1 TO 5
9305   IF P(N) < > 0 THEN SOFAR = SOFAR + 1

9310   NEXT
9320   IF SOFAR < > 4 THEN RETURN
9330   GOTO 8600
```

Finally we look for our default winner using the same SOFAR variable from last time. If we have one, we go to 8600, taking off from where we handled this before. If not, programming continues back to the 10000 traffic director.

FINISHING OFF THE GAME

It's all over now but the shouting. Everyone has gotten cards, the betting is finished—it's time for the showdown.

FIVE-CARD DRAW

```
27000 REM  DECLARATION
27010 GOSUB 47000
27020 FOR X = 1 TO 5
27030 IF P(X) < > 0 THEN 27080
27040 ON X GOSUB 7000,7040,7080,7120,7160
27050 IF X = 5 THEN GOSUB 27100
    : PRINT @LOK,"You've got " + H$(5);
    : GOTO 27080
27060 IF H(X) > 5 THEN PRINT @LOK,H$(X);
27070 IF H(X) < = 5 THEN PRINT @LOK,B$(11)
      ;
27080 NEXT
27090 RETURN
```

There's no need for speed now, which is why this segment is way up there in the programming lines. GOSUB 47000 refers to the formatting lines where the backs of the computer player cards are "turned over" and their true values or CD$(X)s are displayed. We do it sequentially for each player still in the game, and then on the chat lines we print the H$(X) of each hand. If they've got less than jacks (27070), we print "I've got zilch"—B$(11). Otherwise we print the straight H$(X) (27060). For the human player, we're still at a loss, because at this point the computer doesn't know the final value of his hand. We wait until now to find it out on the off chance that the human might have dropped out. Why waste time calculating a nonactive player?

FIVE-CARD DRAW

```
27100 REM  TRANLATE PREVIOUSLY UNTRANSLATE
        D HANDS IN H$(X)+CHECK(X)
27110 REDO = 1
    : GOSUB 130
    : GOSUB 1360
    : REDO = 0
27120 IF H(X) < 6 THEN H$(5) = "DOODLY SQU
        AT"
    : RETURN
27130 IF H(X) = 6 THEN CH(X) = V(X, VAL (
        MID$ (H$(X),2,1)))
    : H$(X) = "PAIR OF " + V$(X, VAL (
        MID$ (H$(X),2,1))) + "S"
```

```
27140 IF H(X) = 7 AND VAL ( MID$ (H$(X),3,
         1)) > VAL ( MID$ (H$(X),5,1))
         THEN CH(X) = V(X, VAL ( MID$ (H$(X
         ),3,1)))
      : H$(X) = "TWO PAIRS "
27150 IF H(X) = 7 AND VAL ( MID$ (H$(X),3,
         1)) < VAL ( MID$ (H$(X),5,1))
         THEN CH(X) = V(X, VAL ( MID$ (H$(X
         ),5,1)))
      : H$(X) = "TWO PAIRS"
27160 IF H(X) = 8 THEN CH(X) = V(X, VAL (
         RIGHT$ (H$(X),1)))
      : H$(X) = "THREE " + V$(X, VAL (
         RIGHT$ (H$(X),1))) + "S"
27180 IF H(X) = 10 THEN GOSUB 2740
27190 IF H(X) = 11 OR H(X) = 12 THEN
         GOSUB 32000
27200 RETURN
```

Here's the starting point for the human REDO. It's much the same as before, with the addition of GOSUB 1360 in line 27110 to get an H(5) for the hand. Then we translate the human holdings into a readable H$(5), much as we did before starting at 2820 (one or two slight differences in the routines account for rewriting them rather than using the same one over again). This done, we now know the H(X) for all the hands as well as the CH(X)s, which we will now analyze.

FIVE-CARD DRAW

```
28000 REM  FIGURE WHO WON
28010 FOR X = 1 TO 5
28020 IF P(X) < > 0 THEN 28070
28030 FOR I = 1 TO 5
28040 IF H(X) < H(I) THEN I = 5
      : FLAG = 1
28050 NEXT
      : IF FLAG = 1 THEN FLAG = 0
      : GOTO 28070
28060 WINNER = X
      : X = 5
```

These lines of programming determine the variable WINNER. We simply run a loop to find out if any H(I) is greater than our H(X) in 28040. If not, we don't necessarily have our winner, but we may. At this point we could have more than one player with the same H(X). All we've determined in 28040 is that no hand values *exceed* this highest H(X), but we've still got to find out if any are equal to it.

FIVE-CARD DRAW

```
28080 Y = 2
      : TIE(1) = WINNER
28090 FOR N = 1 TO 5
28100 IF P(N) < > 0 OR N = WINNER THEN 281
         20
```

```
28110 IF H(N) = H(WINNER) THEN TIE(Y) = N
      : Y = Y + 1
28120 NEXT
28130 IF Y = 2 THEN 28250
```

These lines look for duplicates of the winning high hand, which we now know as H(WINNER). If we do have duplicates, we know that we will have to check the CHs. Here we're looking for how many and which duplicates we have. We set a variable Y as 2, and we initialize the new TIE(1) as WINNER in 28080. We will use Y to count the winners, and we will use TIE(X) to tell us which players they are. In 28110 we look for H(N)s equal to H(WINNER). If we get any, we increment Y for each one. At the end of the loop, if Y still equals 2, we skip down to the winner's circle—we have only one winner. If Y has grown greater than 2, we know we now have Y-1 players with the same H(X) hand value.

FIVE-CARD DRAW

```
28140 FOR X = 1 TO Y - 1
28150 FOR I = 1 TO Y - 1
28160 IF CH(TIE(X)) < CH(TIE(I)) THEN I =
         Y - 1
      : FLAG = 1
28170 NEXT
      : IF FLAG = 1 THEN FLAG = 0
      : GOTO 28185
28180 WINNER = TIE(X)
      : X = 5
28185 NEXT
28190 MULTI = 2
      : MULTI(1) = WINNER
28200 FOR N = 1 TO Y - 1
28210 IF TIE(N) = WINNER THEN 28230
28220 IF CH(TIE(N)) = CH(WINNER) THEN MULT
         I(MULTI) = TIE(N)
      : MULTI = MULTI + 1
28230 NEXT
28240 IF MULTI > 2 THEN 28320
```

This may seem complicated, but it's all very logical, and it's the sort of value-sorting problem that could easily come up in your own programming, so try to follow it as closely as you can.

Using a pair of loops, we now check to see who has the highest CH value among the winning hands. These loops give us the value of the winning CH, but as before, they still don't exclude the possibility of duplicate winners, this time not only with the same H(X) value but also with the same CH(X) value! We test for this in 28190–28230. We initialize yet another variable, MULTI (for MULTIwinner), in addition to the array MULTI(X) (it's okay to have both MULTI and MULTI(X)—the computer will keep them separate, just as it keeps separate H(X) and H$(X)). Then we run a loop to see if there are any duplicate CHs. If so, MULTI will be incremented, and if MULTI is greater than 2 in line 28240, more programming will have to be done to solve the problem.

FIVE-CARD DRAW

```
28250 REM  WINNER'S CIRCLE
28260 ON WINNER GOSUB 46100,46200,46300,46
        400,46500
     : PRINT @LOK,FINGER$ + P$(WINNER) + "
        WIN";
     : IF WINNER < > 5 THEN PRINT "S";
28270 GOSUB 63000
     : R(WINNER) = R(WINNER) + PT
     : PT = 0
28300 GOSUB 7160
     : PRINT @LOK,"Press space bar to conti
        nue " + FINGER$;
28310 C$ = INKEY$
     : IF C$ < > " " THEN 28310
28315 RETURN
```

First let's look at one winner, plain and simple. Here we know that the winner is player number WINNER, hence 28260. In these statements we are printing a FINGER$ in front of the name of the winning player above his cards. Then we give him or her the money in the pot, RETURN, and leave the subroutine. That's when things work out normally. If we have duplicate winners, which is now possible, we do this:

FIVE-CARD DRAW

```
28320 IF TPC = 0 AND H(MULTI(WINNER)) = 7
        THEN 28370
28325 FOR N = 1 TO MULTI - 1
28330 ON MULTI(N) GOSUB 46100,46200,46300,
        46400,46500
     : PRINT @LOK,FINGER$ + P$(MULTI(N)) +
        " WIN";
     : IF MULTI(N) < > 5 THEN PRINT "S"
28340    R(MULTI(N)) = R(MULTI(N)) + CINT
        (PT / (MULTI - 1))
28350 NEXT
28360 GOSUB 63000
     : PT = 0
     : GOSUB 7160
     : PRINT @LOK,"Multiple winners split p
        ot.Press space bar " + FINGER$;
28365 C$ = INKEY$
     : IF C$ < > " " THEN 28365
28366 RETURN
28370 FOR Z = 1 TO 2
28380 FOR I = 1 TO 4
28390 FOR II = 2 TO 5
28400 IF I = II OR V(MULTI(Z),I) = CH(MULT
        I(2)) THEN 28420
28410 IF V(MULTI(Z),I) = V(MULTI(Z),II)
        THEN CH(MULTI(Z)) = V(MULTI(Z),I)
28420 NEXT II,I,Z
```

```
28430 TIE(1) = MULTI(1)
     : TIE(2) = MULTI(2)
     : Y = 3
     : TPC = 1
     : GOTO 28140
```

In our game we're going to invoke a house rule of sorts that will rarely if ever arise. We are going to define multiple winners as what we want multiple winners to be. If both players hold a pair of aces, they will split the pot regardless of the other cards in their hands. The same is true for two players holding flushes with the same high card: They will split the pot regardless of their other holdings. However, from 28370–28430 we run a further check for two-pair hands, seeing whose second pair is higher if the two highest pairs are equal (the variable TPC is a flag for Two-Pair Check). If we do have multiple winners, we run a flash of all their names, and they split the pot (throwing away any loose change—see the CINT in 28340. CINT rounds to the nearest integer, as compared to INT, which always rounds down). Now we're ready to clear the decks and deal another hand.

FIVE-CARD DRAW

```
29000 CLS
29010 FOR X = 1 TO 5
29020 PRINT
29030 IF P(X) = 10 THEN P(X) = 0
29040 IF P(X) < > 0 THEN PRINT P$(X) + " i
        s out!"
     : GOTO 29090
29050 IF R(X) < 1 AND X = 5 THEN FLAG = 1
     : GOTO 29090
29060 IF R(X) < 5 AND X < > 5 THEN PRINT P
        $(X) + " has only " + STR$ (R(X))
        + " dollars."
     : PRINT "Goodbye, " + P$(X)
     : P(X) = 1
     : GOTO 29090
29070 IF X < 5 THEN PRINT P$(X) + " has $"
        + STR$ (R(X))ELSE PRINT "You have
        $";R(5)
29090 NEXT
     : IF FLAG = 1 THEN FLAG = 0
     : GOTO 29320
29110 PRINT
     : PRINT
     : INPUT "Another hand (Y/N) ";C$
29130 IF C$ = "N" THEN PRINT
     : PRINT "See you later!"
     : END
29140 IF C$ < > "Y" THEN GOTO 29110
```

First we CLS. Then we run through each player, displaying his or her bankroll. Then we give the human the option of playing or dropping out, ending if he chooses the latter option.

FIVE-CARD DRAW

```
29150 REM  CLEAR THE DECKS
29160 GOSUB 29170
    : GOTO 29290
29170 FOR I = 1 TO 5
    : B(I) = 0
    : LB(I) = 0
    : IN(I) = 0
    : NEXT
29190 BE = 0
29200 CH$ = ""
29210 FOR I = 1 TO 5
    : CH(I) = 0
    : DISC(I) = 0
    : BF(I) = 0
    : BL(I) = 0
    : DUBCHK$(I) = ""
    : NEXT
29220 FOR I = 1 TO 5
    : F(I) = 0
    : H$(I) = ""
    : H(I) = 0
    : NEXT
29230 NB = 0
    : TPC = 0
29240 FOR I = 0 TO 3
    : NC(I) = 0
    : NEXT
29250 REDO = 0
    : SOFAR = 0
29260 FOR I = 1 TO 5
    : TIE(I) = 0
    : NEXT
29270 WINNER = 0
    : STARTER = 0
29280 RETURN
```

Here we do some more precise deck-clearing, emptying out all the variables we want to empty out. We can't simply use CLEAR, because we need to keep our R(X)s and Dealer (and, in unopened hands, our PT), so we have to go through all of this just to hold one small handful of values. Needless to say, to clear the decks like this, one must know the names of all the variables in the program, which means a bit more tedium on your part. But it's easy enough to do, and it is extremely important if you don't want your game to go screwy. This section is set up as a subroutine because it is used both here and from earlier in the program where there were no opening hands.

FIVE-CARD DRAW

```
29290 D = D + 1
    : IF D = 6 THEN D = 1
29300 IF P(D) < > 0 THEN 29290
29310 RETURN
```

```
29320 PRINT "Sorry, but you're out of mone
          y. That's the end of the game."
29330 END
```

Finally, we increment good old D and RETURN back to the traffic controller if we have the chips. If not, we'll have come to 29320 from 29050, and we'll learn THAT'S THE END OF THE GAME. And believe it or not, it is. The only thing missing is the high-up pauses:

FIVE-CARD DRAW

```
63000 FOR PAUSE = 1 TO 1500
      : NEXT
63010 RETURN
63020 FOR PAUSE = 1 TO 500
      : NEXT
63030 RETURN
```

You might now want to take a week's vacation before proceeding with the next, and final, chapter of the book. I know I'm going to.

12
DEBUGGING THE TOUGH ONES AND OTHER CHOICE TOPICS

DEBUGGING

Debugging is not much of an issue with short, uncomplicated programs. For that matter, debugging is not much of an issue with long, uncomplicated programs. Look at "Space Derelict!": There all we had to do was run the program, plugging in our play inputs until everything looked the way we wanted it to look. But in effect, with the exception of a few small side roads, our adventure module consists primarily of only one major routine, the conversation with the computer. All the side road sections are small enough and elementary enough to lend themselves to almost instantaneous debugging—either they run or they don't. With the conversation segment, we have the further virtue of using only a handful of variables, with S$, S2$, and C$ taking up the lion's share of them. And what's best of all, these variables are entirely dependent on the player inputs—the C$ that the player plugs in—so to debug, all we have to do is type in every combination of C$ input we can think of and then make the appropriate corrections when an input doesn't work. Although some of those corrections can be complicated, the process itself is relatively straightforward. You might spend a bit of time debugging an adventure, but you'll seldom find yourself stumped on a logic poser. Strategy games, however, will not be so easy to straighten out, so be prepared for interminable debugging sessions easily as long as the original creation of the program.

The main reason for the difficulty in debugging this sort of program is the intrinsic complexity of the program itself. Even if many of the subroutines are not particularly perplexing in their design, they are nonetheless long and unruly. And some of the subroutines, especially the hand-figuring and the subsequent draw and refiguring in ''Five-Card Draw,'' are extremely murky. There's so much going on in these segments that there's just no way of debugging by plodding through it, as we did with ''Space Derelict!'': just running it until we get an error, correcting the error, and running it some more. What's worse, we don't have that great time-cutting option of the saved game that we had in our adventure. And worst of all, whereas in our adventure we controlled most of our variables from the keyboard via player inputs, in our strategy game most of the variables are generated by the program itself. The CD$(X)s, the H(X) and H$(X)s, the betting variables—with the exception of a handful of pieces relevant to the human's own hand that are input by the player, all of these are controlled by the program. And to debug successfully, we have to make sure that all of them are controlled accurately.

In addition to getting our program to play correctly, we will also be thinking about flow speed in our debugging stage. When laying down a program at the beginning, it's best to lay it out from line 10 to line whatever without worrying about what subroutine falls where in terms of internal access speed. Then, when you have it all on disk, Renumber it, starting at 10000 with line increments of 10. Now, by deleting around sections you want to move and then using MERGE, you can shift the slow-moving segments down to the lower reaches below 10000 and keep the ones where time doesn't matter on the high-numbered lines. This is how I worked with the poker program. First I wrote it from start to finish, then I renumbered starting at 10000, adding that line 10 (GOTO 10000) as my first statement. Then, as I debugged, I made my decisions about which subroutines needed prime real estate and which ones didn't, and I adjusted accordingly using MERGE. Let's say you wanted to move 12000–13000 down to 2000. First delete all program lines except 12000–13000. Then RENUM 2000 and save these lines as TEMPS. LOAD ''POKER'' and delete lines 12000–13000 and then MERGE ''TEMPS'', and the job will be done. It is not impossible to preplan the placements of the fast and the slow segments, but it isn't really necessary: RENUM and MERGE do a lot of this work for us after the fact, when we can clearly analyze the speed of a routine in action, rather than making us worry about the fact beforehand.

As I said, the worst thing about strategy games is the program-generated variables. You'll recall when we discussed structured programming way back when that one of the reasons for using this method was to aid in debugging. Now you'll see why, when we use our modular subroutines as independent units, debugging one at a time, substituting our own keyboard-generated variables for the program-generated ones. We're going to take one subroutine at a time, debug it, then go on to the next subroutine. And we'll start at the beginning, with line 10000. Our debugging style for ''Five-Card Draw,'' moreover, will be appropriate for any other strategy game.

Subroutines 60000 and 50000 don't require much sweat, because neither of them really does anything all that earthshattering. Here we can simply RUN and see what happens. Since there are no conditionals in these lines, every line will be processed, and syntax errors will immediately pop out at us for correction. That done, we get to line 6000, the shuffle and ante. Here we're going to be looking for two things: first, that we get 40 different cards, and second, that our R(X) bankroll values are decremented correctly. So we're going to plug in a new statement such as this:

```
10025 GOTO 11000
```

which will take us out of our normal flow altogether after the shuffle and ante, and then we'll add some new lines such as this:

```
11000 FOR X = 1 TO 39 STEP 2: PRINT C$(X) + S$(X), C$(X+1) +
S$(X+1): NEXT
11010 INPUT C$
11020 FOR X = 1 TO 5: PRINT R(X): NEXT
11030 END
```

The first thing this new section of programming will do—and note that this is not part of the actual program but only a temporary debugging tactic—is print out our 40 and values and suits, i.e., CD$(X) without the rectangle. Line 11000 prints two CD$s on a line; the comma between the concatenations is a screen-formatting command that causes data on the right side of the comma to print one TAB position over from the material on the left side. We do this here to get all our 40 variables on the screen at once. We can look at them now and make sure there are no duplicates. If there are, back to the drawing board. If there aren't, hit a dummy input to see what the R(X)s are. They should be 99 (the original 100 − 1); if so, great; if not, correct. And on from there.

We're going to follow this practice of creating short debugging routines in one way or another for just about every section of the program. In some cases we'll go even further: Not only will we analyze data after the fact, but we'll actually plug in the data first. You can see from this just what we're doing. We're not going to look at the program as a whole. We're just taking it one step at a time, making sure each step works, and then going on to the next step. If we can get all the pieces to work individually, then the whole will take care of itself. After we're finished with each piece we'll delete our debugging programming and save the whole with the latest finished section up and running until we have all the sections ready to go.

Our screen formatting for the deal, to which we branch from 10030, does not have to be fixed in this fashion with a debugging routine. When we get to this subroutine either the blank cards and so forth will end up where they belong or they won't—the results on the screen will be obvious. So we can just run this, see how they fall, and act accordingly. Most likely, any screen formatting, no matter how well planned, won't be perfect the first time you run it, but you'll most likely find one or two elementary mathematical errors that will be easy to correct, and then you'll have what you want.

The next subroutine is the hand-figuring, and we're going to take this segment entirely on its own. Here we have to input our own card values to make sure we get correct H(X) and H$(X)s at the other end. Moreover, we want to do this in a controlled way so we'll be able to check every possibility. We're going to bypass the rest of the program and begin like this:

```
10 DIM C(25), C$(25), S(25), S$(25), CD$(25): GOTO 11000
11000 FOR N = 1 TO 25
11010 PRINT "C(N) - C$(N) - S(N) - S$(N)"
11020 PRINT N: INPUT C(N): INPUT C$(N): INPUT S(N): INPUT S$(N)
11030 NEXT
11040 GOTO 10040
```

This will allow us to type in each individual C, C$, S, and S$ ourselves, which means we can plan them in advance. We can start with each hand possibility and plug in the right cards five at a time for all five hands and then send these values into the hand-figuring routine. The best way to make use of this is to go along with the way this figuring subroutine at line 90 is actually designed: First we might try plugging in various pairs, then we might try different two-pair combinations, threes of a kind, and so forth, all the way through the insidious three flushes and inside straights. Here's how we catch it at the other end:

```
10045 GOTO 12000
12000 FOR X = 1 TO 5: PRINT H(X),H$(X): NEXT
12010 END
```

Here we see the resulting H(X) and H$(X)s. Since we know what we plugged in, we can compare the hands to the end results. If we get the right H's, we're okay; if we get the wrong ones, we have to figure out why. The important thing is that we are entirely in command of the process, which means that we get (or at least ought to get) exactly what we want. But we have to be very careful to input *every* possible combination. You might put in a hand like this

QUEEN ♤
QUEEN ♡
FOUR ♧
TWO ♢
NINE ♤

and get the result P12J, which is correct. But this does not necessarily mean *all* pairs will be so collected. At this point in debugging, for all we know the P12 might work and the P35 might not. You have to plug in *every* possibility (or at least thoroughly representative samples thereof) until you're absolutely certain they all work. For instance, I mentioned the DUBCHK$(X) problem earlier, where the way I had figured four straights I could be missing the boat if the first two cards in the hand happened to fall the wrong way. I didn't think of this originally and didn't discover it until I had plugged quite a few four straights into this debug programming and accidentally had come upon this hidden problem. With some four straights, my H's just weren't reading right. A little agony, and DUBCHK$(X) came into existence. And by the way, you can see now why the various straight hands got the H$(X) names they did: By putting the correct numbers of each card in order in the H$(X), I was able to make sure not only that I had the straight I was looking for but also that the computer and I agreed about which cards were which that were making up that straight.

The first round of betting, from 10050, was debugged from the program—i.e., I watched it run on its own, with me following the progress of the info display in the corner, keeping notes, making sure that everything went as it was supposed to. Then I'd break into the program to make sure the R(X)s were still where they were supposed to be after the betting was concluded (when you break into a running program, the variable values remain intact—until you make changes in any of the programming lines). Needless to say, when you're debugging you'll need a nice pad of paper in front of you to keep track of what you're doing.

To debug the draw segment I let the computer do the first part: Why should I bother inputting all those card values after I knew the program was debugged up to that point? I simply added some new lines like this:

```
10065 CLS: FOR N = 1 TO 5: PRINT H$(X): NEXT
10066 INPUT C$
```

and then let the subroutine do its stuff, followed by a new debug program at 11000 like this:

```
10075 GOTO 11000
11000 FOR X = 26 TO 40: PRINT C$(X) + S$(X): NEXT
11010 FOR X = 1 TO 5: PRINT H$(X): NEXT
```

This would give me the H$(X)s going in, which I would write down, and then I would get a list of the discards available in order at 11000, followed by a tally of the H$(X)s going out of the draw routine. Knowing what each player had initially, I could tell now whether the discards had been correctly drawn and whether the hands had been amended accordingly. As for the rest of the game, it was pretty much debugged from the actual program, with an occasional break-in to read specific values to make sure everything was on the up and up.

I've simplified a bit here, because every now and then I'd design a debug program a lot more complicated than these, giving me all sorts of information about the hands all at the same time, or else I'd go a lot more deeply into one segment of a given subroutine, hellbent on a very specific task requiring very specific debugging. But the section above is meant as a guide, not a bible, and you should be able to get from it the gist of what we're talking about here. When you're debugging a complicated program, you have to take that program apart and examine every nut and bolt, and this is the best way to do it. Sometimes you'll have to go even deeper, but occasionally you'll only have to scratch the surface. But be prepared to spend a lot of time at this stage of the program. Everything we said in Part One of this book about debugging is doubly true here, and there are no shortcuts in this part of the forest.

OTHER STRATEGY GAMES

My expressed purpose in Part Two of this book has been to provide you with a complete and fairly complicated strategy game in the hopes that seeing how the problems were solved in it would help you solve problems in any other strategy game you might design yourself. This means that short of designing a different brand of poker, in which case all you would have to do is co-opt this program, changing a few small parts, you're pretty much on your own. But we can still talk about a few ideas for some other games and see how what we've discussed so far can be used directly in designing those programs. And the best place to start is with other card games.

Possibly the easiest game you can design is blackjack, because there's no need to incorporate any strategy for the computer. In casino blackjack the dealer must rigidly adhere to the rule of taking a card on 16 and standing on 17 (although some casinos allow the dealer to hit a "soft" 17—i.e., a hand where an ace is counted as 11, as in ACE CLUB, SIX SPADE). All you have to do is make the computer the dealer and provide a strict accounting of the values of the hands to figure out what everyone is doing. Then you figure out the winner, pay out or take in the money, and deal another hand. In an elementary program, one player vs. the computer/dealer, the only real programming complication would be accounting for aces, which can have a value of either 1 or 11, at the discretion of the player. What you might do is default aces as 11. Then

```
H(X) = V(X,1) + V(X,2)
```

to give you the original H(X) of a hand. Then with each hit:

```
H(X) = H(X) + V(X,N): IF V(X,N) = 11 AND H(X) > 21 THEN H(X)
= H(X) - 10
```

This will give you the highest possible H(X) under 21. Aside from this, all you have to do in evaluating the hands is add up the V(X,N) values, calling a bust when it goes over 21. Player cards would be dealt as the player requests them: whether the dealer takes any additional cards would be determined by the value of the dealer's hand.

By now you may know me well enough to guess that I personally would not be satisfied with a blackjack game where just one player sat down and played against the computer, and the one who got closest to 21 would win. Too simple, and not a real simulation of what casino blackjack is all about. It would be essential that the play allow options for doubling, pair splitting, surrender, and insurance, at the very least (these are all betting strategies, variations of which are usually available at the casino tables). There's no fun to betting unless you're allowed to make all the bets. Second, in real casino play the house uses anywhere from one to eight decks at a time; this too should be an option, perhaps chosen at the beginning of the game. Needless to say, such a long shuffle of many decks would require some hiding to prevent slowness. Third, the player should be allowed to play more than one hand at a time, or perhaps the program would allow for more than one player so that up to seven people could play at once against the machine. Again, this is just like real life, and the more computer games of this nature resemble real life, the better they are. These additions to the program would most definitely make it that much sharper and that much more fun.

There is one last facet of blackjack, which I address to students of the game. You may not know it, but there is a large body of literature available on the game of blackjack. The reason for this is that for the diligent player there are a number of systems that will beat the casino odds. The percentage in the player's favor is somewhere around 1 percent, but over the long term any percentage will earn money. Unfortunately, to be able to play and get these odds on your side, you must not only play your cards with exceptional precision, but you also must ''count'' the values of the cards as they are dealt. Precision play isn't too hard, because all you have to do is memorize a few dozen hand combinations and always play according to those combinations. A man named Edward O. Thorp originally introduced these combinations to the world in a best-selling book called *Beat the Dealer*. These combinations are called Basic Strategy, and they will entirely determine whether you will stand or draw a card, based on the total value of your hand against the up card the dealer is showing. The other side of the coin is the counting of values, and there are numerous sytems for this. Usually this requires keeping track of all the aces and 10-value cards (10 through king) as they are dealt in ratio to all the other cards. This means you will have a running tally in your head something like +2 or −1 or 0 or −4. This number will tell you whether to increase or decrease your bet and by how much. For more on both Basic Strategy and counting, any of those books on the blackjack shelf will get you started in the right direction.

Anyhow, wouldn't it be interesting to program some of this information into your own game? You could have a computer-generated player who always bets Basic Strategy, and you could see for yourself how he does over the long term. You could even have this computer-generated player make his bets on the basis of a counting system. Or you could program a running tally of the count to be displayed on the screen as a tutorial. The possibilities are fascinating, and they are not outside the realm of the Basic programmer with a real interest in blackjack.

On the other side of the spectrum from blackjack, in both programming and play, is the game of bridge, considered by many to be the ultimate card game. Right at the top, the biggest problem you might have in tackling bridge is the size of the program—you could probably eat up 7 or 8 megabytes if you really got carried away with it. I have seen one bridge program in the public domain that is actually two CHAINed programs, one to deal and bid, the other to play. Not a bad idea to consider yourself. I've never looked inside this Basic program, but I have some of my own ideas on how I would do it if I were to attack such a monster.

The bidding side of it wouldn't be too hard, regardless of the bidding system used. Bridge bidding is fairly rigid; in the good old Standard American I used to play in college, for instance, you would only bid "one no trump" with an evenly distributed hand, 16–18 points. If you had even distribution and 20 points, one no trump was out of the question. And so forth. Every hand, any bidding system, has specific predetermined right and wrong bids; in your bidding segment you would define your hands both by point count, an H(X) value perhaps, and distribution, perhaps a D(X,Y), where X is the player and Y is the suit from 1 to 4, and the value of D(X,Y) would be the number of cards for that player in that suit. Not bad, and it would answer just about everything you need to know. All you would have to do is program an algorithm where, given all the possible H(X) and D(X,Y) values, you would instantly come up with the appropriate bid. Remember, the computer would have to bid for three hands, while the human player would be responsible for only one of them.

The hardest part about designing bridge would be the algorithms of play. While the human played his own hand and the dummy, you would have to handle the two opponents. Here your program's success would rest very much on the two essentials of strategy-game design: your complete knowledge of the best strategy combined with your ability to translate that strategy into computer play. You would want to put in as much as possible: the ability for the opponents to play their leads in response to the bidding, in response to the dummy, and even in response to each other. There are so many possibilities for all of these that you can easily see why, as of this date, the critics say that the ultimate bridge program has yet to be written (which is odd, because they've done some amazing things with chess—

maybe there just haven't been any grand master bridge players who have gotten around to buying themselves a computer yet). Maybe you could be the one to do it. Why not?

There are dozens of other games out there waiting to be programmed. And if you're an avid player of any of them, the successful design of a challenging computer version can be an incredible satisfaction. Backgammon? Othello? Go? Sure, these are all available in store-bought programs, but although they may be better than anything you could ever design yourself (and that's a very big maybe—we're nowhere near the end of the strategy-game-software spectrum at this early date), none of them can provide the thrill of doing it yourself and doing it well. For me, the ultimate challenge is Monopoly—the speed of the computer to make the moves, nice graphics, but most of all, killer algorithms to control the wheeling and dealing segments of the game. It's definitely on my list of projects to come (you can put it on yours, too, if you like; the more the merrier). I get to play the real game only at Christmastime, after using the holidays as an excuse to gather enough players at one time to get a game up. But if I programmed it, I could be playing right now. . . .

THE ULTIMATE STRATEGY GAME

The ultimate strategy game? Is there really such an animal? In fact, there is: It's the strategy game you invent yourself, a completely original computer game entirely of your own design. It's not a simulation of any other game, or a new version of some other game. It's new, it's exciting, it's yours. And it can be the most satisfying programming you'll ever do.

The idea of inventing your own game is very daunting. It's not the same as creating an adventure, because even though any adventure you design will be original in plot, it will still be an adventure, a new installment in an already-existing genre. But creating a new strategy game means introducing something that never existed before. Once upon a time chess, checkers, Go, Monopoly, bridge, backgammon—you name it—did not exist. Someone had to invent them. Usually that someone relied on concepts that had come before (protochess games seem to go back almost to the beginning of recorded history, for instance, and there were board games on the market long before Monopoly came out), but nonetheless some inventor looked around, and with the ideas of the past to guide him, developed something new and lasting. Am I making this all sound too grandiose and overdramatic? Perhaps, but to invent a new game that's fun to play is no mean feat and should be relegated the respect due such an undertaking.

With the advent of the personal computer came a whole bunch of new, unique games designed to take advantage of the computer's unusual gaming personality. The adventure game and the arcade game, of course, are the most original of these, in that they did not exist before computers were there to play them, and without computers (or at least computer chips) they could not exist at all. And more to the point, they have proven their durability against the criterion of sheer fun. But in the earliest years (the Dark Ages of the mid-to-late seventies), many other new games were sent down the pike as well, if not necessarily all that exciting to play, at least original and unique to the computer. One genre that comes to mind is what I think of as the king and the wheat—the player takes the role of the leader of a country, and against various adversities must make sure that the gross national product of the domain exceeds the national debt (our most recent presidents could learn something from this sort of game). Then there's the genre I call the impenetrable war module, where you control one faction and the computer controls the rest, and your job is to figure out the rules before you lose the game. Usually these confusing games take place out in the galaxy somewhere and have as many possible moves as there are keys on the computer. All of these games are in the public domain and are not hard to come by. And for the most part, all these games have one thing in common: They aren't much fun to play. As exercises in programming they might have been satisfying to their creators (something I highly endorse), and as finished programs they may be edifying to other programmers (again, nothing to sneeze at), but as games per se they leave a lot to be desired: Five minutes later you're ready for another one. To be honest, I too have tried to create one of these talky, rule-heavy games, which after a few months I finally abandoned because when a friend of mine tried out the first stage that had already been programmed he fell asleep on the keyboard.

So what is the key to creating a really good new game? The traditional game industry, those people who market new board games and whatnot, has had a bylaw almost as old as the industry itself, and that bylaw has already entered the parlance of computer game people: easy to learn, difficult to master. There are axioms and corollaries to this law, but they all hearken back to that one dynamic: easy to learn, difficult to master. It's the litmus test that any successful game can pass with flying colors. Look at poker, for instance. You get a given number of cards, and you try to match as many as you can, either by suit or by value. That's all there is to it. Checkers—you move your pieces one at a time to the opposite end of the board, jumping the other player's pieces along the way. Ditto Chinese checkers. Backgammon—you try to move your pieces to the opposite end of the board before the other player does (it almost sounds like checkers when you put it that way). Even chess has only six different pieces, each with one or two rigidly defined moves on a static playing field; all you have to do is trap one particular piece of your opponent's before he traps that same piece of yours. Nothing to it, when you boil it down to its simplest elements. All of these games, even chess, can be learned in a few minutes. Most of them can even be played in a few minutes. And all of them are endlessly fascinating and have withstood the test of time. That's the sort of thing you want to accomplish with your own homemade game. Easy to learn, difficult to master.

The best way to develop this sort of game is on paper. After you get a few of the basic elements set in your mind, forget about the computer for a while and see if the game will play. Cut out some cardboard pieces and play it, adjusting, changing, fixing as the game develops. Try different pieces, different moves, different playing fields. Add new rules, modify old ones, simplify as much as possible. Get it down to something that really plays and then learn to play it, because your program is going to have to know perfect strategy to provide the player with a good opponent. Only after you've done all this are you ready to lay down your first programming statement.

One thing to avoid at all costs is randomness. Short of something like a deal of the cards, with its almost infinite variations, random factors do nothing but detract from a strategy game. The more random factors there are, the less strategy there can be, by definition.

A nice thing to put into your game (whether your creation or otherwise) is a set of play levels, from the easiest to the hardest. This gives the human player a chance to start easy and graduate to the more difficult stuff. And it's not that difficult to do. Simply provide an input early on for the player to choose his option and then set a level flag in the program accordingly. Then use that flag to branch to strategic/tactical subroutines in the program, one subroutine for each level. Your algorithms in each subroutine would get progressively more sophisticated as you go along, without necessarily changing their basic design. In a way we did something very much like this in our poker program by creating four players with different styles of play. Most of the program plays the same regardless of which player we're talking about; it is only in those special lines where we assigned the DF(X), RF(X), and BF(X) values, based on the content of the hands, that anything changed all that much. Imagine that we had designed this game where the human played only against one computer opponent at a time. In that case we would have given the human four options, play levels 1 to 4. At level 1 we'd have given him Marvin; level 2, Betsy; level 3, Walter; and level 4, Gerry. Not all so difficult, when you think of it that way. Indeed, in some games level changes might require more than just the tinkering of three variables, but trying to keep level changes to the minimum number of variables would have to be one of your first priorities. But if you needed other changes along the way, your level flags would do the job for you. Let's say that your level flag is the variable LF. Whenever you need to differentiate play on the basis of level you would simply program ON LF GOTO or IF LF > 3 THEN or somesuch. It can and should be done if you want your game to be all it can be.

CONCLUSION

''Mother of mercy, is this the end of Rico?'' I'm afraid it is, except for the Glossary. We've come a long way in these few hundred pages, and we've covered a lot of territory. We've gone over a couple of megaprograms that eat up just about all the memory we have there for the eating, and we've talked about everything I could think of to help you program similar—perhaps even better—programs yourself. Some of it has been cockamamy, my own personal style, and some of it has been the wisdom of the ages. A lot of it has been somewhere in between. And now I'm at a loss to tie it all up in one neat little package. I'd like to end on some profound note, but nothing even remotely profound seems to come to mind. I guess I've just plain and simple shot my wad.

My final word, in the time-honored tradition of computer people everywhere, is this: If you have any comments, questions, fixes, or changes in the programs you'd like to share, improvements perhaps for future versions, anything at all, let me know. The publisher's address is on the title page of this book, and even though I'm getting out of town on the next stage, they will always know how to reach me and will forward any correspondence.

Cheers!

GLOSSARY

adventure game. A computer game where a player (or players) tries to solve a complex puzzle by traveling through the game universe and successfully manipulating the objects therein. In this book we differentiate between treasure hunts, in which the player tries to amass riches, and situational adventures, where the player has to perform a certain complicated act to win the game.
algorithm. A formula or series of formulas or a section of programming that boils down a complicated computing process into one tight quantification of data.
ALL. Command used with CHAIN to denote the passing of ALL variable values from program to program. A COMMON statement is always recommended as a CHAIN pass-along fail-safe, ALLs notwithstanding.
AND. A Boolean operator. In an AND statement, both sides of the AND must be true for the whole condition to be considered true. *Compare* OR.
array. The method by which one variable name can be extended to cover numerous situations; arrays are usually used when variables are related—for instance, door 1 through 17 represented as D(1) through D(17). This example is a simple array. You could index more complicated arrays such as V(1,2) or S(3,4), which we do in this book, or even something like ZI(4,2,5,3,10,24), although it is unlikely you would be able to make any sense out of it afterward. Variables can be used instead of numbers within those parentheses; the maximum number of values you can assign to your array is called the dimension (*q.v.*).
ASC. As in ASC(''B''), this function gives you the ASCII (see below) value of the character in parentheses. In this case ASC(''B'')=66.
ASCII. All the characters on the keyboard, including Control characters,

Escape, smiley faces, etc., are understood by the computer only as bytes, values from 0 to 255. These values are the ASCII character codes. Depending on your particular machine, the TRS-80 uses all 256 of them, in some cases with alternate characters as well; see your manual appendices for more information.

assembler. A program that translates mnemonic commands into hexadecimal machine-language programs.

AUTO. As in AUTOmatic numbering, writes your line numbers for you as you program. Can be set to start at any number you want, using any increment (within reason, of course). Format is AUTO X,Y: X for starting line, Y for increment, both being optional with defaults of 10. A BREAK turns AUTO off.

binary number. Binary numbers are based on the number 2 and are the native vocabulary of a digital computer. The Z-80 processor in the TRS-80 traffics in 8-bit information—i.e., binary numbers 8 characters long.

bit. *See* byte.

Boolean logic. The correct manipulation of AND and OR operators and all combinations thereof.

branch. Branching is the act of going from one part of a program to another, through either a GOTO or a GOSUB.

BREAK. This key stops a running program, disables AUTO, stops a LIST, etc. When you break into a program all the variables remain live—i.e., they all hold their most recent values—until you change something, at which point the variables are cleared. A CONT will CONTinue execution of a broken program.

buffer. An area set aside in memory for holding disk file data.

bug. An error of logic that causes your program to run awry. A bug should be considered the next generation away from simple syntax errors or other, similar mistakes in programming that the Basic Interpreter will catch on its own. Bugs are the errors you'll have to catch yourself.

byte. A byte consists of 8 bits, and each bit is a binary number, 0 or 1. So a byte would look like this: 00011001. However, when your TRS-80 talks to you it usually speaks decimal, not binary, so a byte becomes any value from 0 to 255 (or from 00000000 to 11111111, the maximum byte). This information plays no part in the plot of this book.

CAPS. Works like the lock key on a regular typewriter. TRSDOS starts you out in all lower case. Hit CAPS, and you get all upper case. Hit it again and you get all lower case. And so forth. Upper or lower or a mixture of both, either in commands or variables, are all read as equivalents by TRSDOS.

carriage return. The end of a program line or an INPUT, so noted by pressing the Enter key.

CHAIN. Command that allows you to go from one program to another with the option of passing the values of the variables between programs. A COMMON statement is either mandatory or else recommended as an adjunct to CHAIN if variables are passed.

CHR$. CHR$(X) takes the character in the parentheses and returns its ASCII (*q.v.*) value.

CINT. Rounds numbers up or down to nearest integer value. *Compare* INT.

CLEAR. The command to reset all variables within a program to 0. Can also be used to set the high point in memory (very advanced) and set stack space within memory (usually unnecessary).
CLOSE. The command to put away a disk file for another day.
CLS. A command that clears the screen of all text, placing the cursor at the top left of the screen.
colon (:). Not a part of human anatomy but an important punctuation in Basic statements. A colon can separate multiple statements on one line. But note that if any statement on the line is a conditional (IF . . . THEN), processing will continue past the colon only if the condition is met.
command. Anything you tell the computer to do immediately, also known as direct or immediate command. *Compare* statement.
compiler. A program that translates a high-level language program into machine language; it is the compiled program, and not the high-level one, that is eventually run on the computer.
complication. In an adventure game, a complication is anything the player has to figure out to proceed. The more complications in an adventure, the better.
concatenation. The joining together of two disparate items. Multiple strings are concatenated by + symbols (A$+B$). A string is concatenated to a number by a semicolon (INPUT ''HOW MUCH'';X). Variations on these themes abound.
conditional. A statement that begins with IF. Whatever is between the IF and the THEN is the condition.
CONT. *See* Break.
copy protection. The process of scrambling data on a disk to keep people from copying the data.
counter. A variable in a program that has no particular meaning of its own. A counter variable is used to move through loops where the variable itself is of no importance—e.g., FOR X=1 TO 5: IF IMPORTANTVARIABLE = X THEN Z = 10: NEXT. X is the counter in this statement.
cowcatcher. Any programming that picks up and acts on all the conditions already excluded by the previous programming. This could be anything from an error trap (when you prompt for a ''Y'' or an ''N'' and the user inputs ''W'') to a conversational response in an adventure.
CPU. Central Processing Unit. The real name for the thing under your TRS-80 keyboard that does the computing.
cursor. The blinking white line that tells you where within the text window the computer thinks you are. As a rule, the computer will always be right about this. PRINT @ commands can control placement of the cursor.
DATA. In a program, DATA lines contain just that, a collection of data to be READ (*q.v.*) by the program when the time comes. DATA lines can be placed anywhere within a program without affecting processing, and string data need not be so denoted by quotation marks.
DATE$. Today's date, input when you turn on your machine. This value can be manipulated within programs.
debugging. The process of going through a program and taking out the bugs. This can take anywhere from 50 to 1000 percent of the original time spent programming.
default. The normal setting or value for a machine address, variable, or

operation. For example, undimensioned arrays default to 11 values—i.e. X(10).
DELETE. Command to erase program lines.
DIM. The command for setting the extent of your arrays. The format is DIM X(27), for example. With complicated arrays the format could be DIM X(4,10), or DIM X(9,9,9), and so forth. Dimensions of arrays default at 11 values (e.g., DIM X(10), since the zero value is counted), so you don't have to dimension arrays of this size or less. One-line statements such as
DIM X(24), Y(19), Z(4,4)
are perfectly acceptable time- and space-savers.
dimension. *See* DIM.
direct-access files. Disk files that can be accessed from any field. *Compare* sequential-access files.
disk file. Either sequential or direct-access, disk files are collections of data physically stored on a disk.
EDIT. The function for editing lines in a program. Much too complicated a process to go into here. The EDIT mode is fully explained in your manuals. This is one of the very attractive features of TRS-80's Basic.
ELSE. An extension of the IF . . . THEN statement. ELSE allows you further to modify conditionals past the one operation of IF . . . THEN.
END. Command to end a program.
field. A piece of data in a disk file.
flag. A variable for marking the existence of certain conditions within a program. Flags are usually either on or off, but they can be as complicated as you want to make them.
flow. The course of a program; a flow chart is a map of this course.
FOR . . . NEXT. A FOR . . . NEXT loop allows you to do the same thing over and over again, substituting different values. FOR . . . NEXTs can go either up or down, depending on the STEP value (the STEP default is 1). So you could say FOR X=1 TO 100 STEP 10 or you could say FOR MAMBO=100 TO 25 STEP -25 or simply FOR ZSAZSA=1 TO 10—it's all the same to the computer. There is much discussion of FOR . . . NEXT throughout this book; its complications are interesting and many.
GOSUB. A branching command to a subroutine—i.e., GO SUBroutine. At the end of the subroutine the command RETURN will send processing to the command immediately following the original GOSUB. *Compare* GOTO.
GOTO. The branching command from which you never return. A command GOTO (SOMENUMBER) will take you to that line number, and processing will continue from that spot. *Compare* GOSUB.
hexadecimal. A numeric system based on the number 16, with values ranging from 0 to F, 10 to 1F, etc. Used because 8-bit processors such as the TRS-80 have 16 fingers, 8 on each hand.
high-level language. A computer language that makes fair sense to humans and is easy to use but that needs some translation before the machine can understand it. Basic is a high-level language.
housekeeping. *See* initialization.
initialization. The process within a program whereby the program variables are defined and, if necessary, set.

INKEY$. When this function appears in a program, the computer is requesting—but not waiting for—the entry of a keyboard character. INKEY$ allows you to INPUT single characters without having to press Enter. To get INKEY$ to work you usually need to put it in a loop: the nice thing about this loop is that it can do anything you want while you're waiting for the keyboard input.
INPUT. The command requesting from the user either string or numeric input, as in INPUT X, INPUT "WHAT'S YOUR NAME";N$, INPUT ZITI$. Processing pauses until information is typed in, followed by an Enter. *Compare* INKEY$.
INT. This command, followed by a variable in parentheses—e.g., INT(X)—rounds down the value of that variable to an integer. So INT(X) where X = 2.66 would be 2. *Compare* CINT.
Interpreter. In Basic, the Interpreter takes each program line as it is processed and translates it into machine-code operations the computer can understand. *Compare* compiler.
inventory. The sum of the items that a player puppet is carrying on his person in an adventure.
kilobyte. One thousand bytes.
LEFT$. Expressed as LEFT$(X$,N), where X$ is some string and N is the number of letters you want counting off from the left.
LEN. As in LEN(MY$), a measure the length of a string.
LINE INPUT. Command that allows us to INPUT anything we want from the keyboard, including commas, something we can't do with plain old INPUT.
LIST. The command to list a program.
LOAD. The command whereby programs are read off the disk and placed into the machine memory, but not run.
logical exclusion. In a program, certain conditions exist at any given time. As your programming lines test for more than one of these conditions, once you've eliminated all the possibilities but one, you don't have to test for that one as well. That is, if X can have only a value of either 1 or 2, it is unnecessary to program like this:
```
40 IF X = 1 THEN 200
50 IF X = 2 THEN 300
```
If X doesn't equal 1 it *has* to equal 2. When you start combining multiple conditionals on one line, this kind of clearheaded thinking becomes very important.
loop. The process by which you keep processing through one or more lines in a program. A limited loop would be a FOR . . . NEXT; an infinite loop would be any form of this:
```
10 GOTO 10
```
Our adventure module, long as it is, is a 99 percent infinite loop, the only exits being winning or losing.
low-level language. A computer language difficult for humans but easy for the computer to understand. Machine language, a series of hexadecimal numbers, is about as low as we can go, using an assembler to get us there.
megabyte. One million bytes.
MEM. Type in PRINT MEM and your computer will tell you how much space in bytes is available to your Basic program.

MERGE. Command that allows us to overlay one program onto another one, or literally to merge two programs. If any line numbers are identical in the two programs, the lines from the new program will supersede the old. MERGE works only on ASCII files and can be combined with the CHAIN command.
MID$. Either MID$(X$,4,1) or MID$(X$,9). In the first case we would go to the middle of string X$ and count off one character starting at the fourth character. In the second case we would start at the ninth character and count off to the end.
monitor. The video screen.
nesting. Putting small loops inside big loops; e.g.,

```
FOR X = 1 TO 5
FOR Y = 10 TO 20
IF X = Y THEN ? "INCREDIBLE!"
NEXT Y,X
```

You can also nest GOSUBs within GOSUBs.
NEXT. *See* FOR . . . NEXT.
null string (""). A string of zero length, a string with nothing in it.
ON. A command that allows multiple branches depending on the value of a given variable. As in ON X GOTO or ON X GOSUB. The ON command will measure the value of X (or whatever); processing will continue on the Xth line referenced after the GOTO or GOSUB. If there is no matching Xth line, processing will simply continue at the next command following the ON statement.
OPEN. The command to find and commence reading or writing a disk file.
OR. A Boolean operator in which if one condition on either side of the OR is true, regardless of whether the other side is false, the entire statement is regarded as true. *Compare* AND.
parse. In an adventure conversation segment, to parse means to break down the player's input into its component parts.
phase. A series of complications (*q.v.*) within an adventure.
pointer. A value in memory that the computer uses to keep track of FOR . . . NEXTs, GOSUB . . . RETURNs, etc.
POKE. The command whereby you can put in specific values at specific addresses in memory.
PRINT. PRINT statements transmit information wherever directed, either to the screen, a printer, or any other output device you designate. More importantly, the symbol ? on the keyboard will translate into the word PRINT, so instead of always typing in five letters you can hit one symbol.
PRINT@. Same as PRINT, but at a specific location on the monitor grid.
prompt. A prompt on the screen is that bit of business that tells the user what to do and where to do it. The blinking cursor following INPUT statements is the usual form of this.
puppet. The "character" that the player moves through the game universe in an adventure.
RAM (Random Access Memory). The free memory in your machine available to you for programming. Some parts of RAM are already taken, in fact, by the video monitor, TRSDOS, program pointers, etc. *Compare* ROM.
RANDOM. Seeds a new random number.
READ. The collector of data from DATA lines.

REM. The command that allows you to talk to yourself within a program. REM statements are ignored by the computer, so anything you write in after you type REM is entirely between you and yourself. REMs are useful for annotating complicated ideas, indicating starting points of subroutines, and slowing down programs. Experience alone will determine how much of a REMer you're cut out to be.
RENUM. RENUMber, as in RENUM X,Y,Z, with the variables optional. X is the number of the first new line, Y is the number of the line within the program at which to begin numbering, and Z is the increment between line numbers. Defaults are 10, the first line of your program, and 10, respectively.
RESTORE. After you READ the data in a DATA line, that information is no longer available to you unless a RESTORE command says otherwise. RESTORE places the pointer back at the first piece of data.
RESUME. The command that internally continues a program from where we left off if we've changed a line.
RETURN. At the end of a GOSUB, a RETURN sends processing to the command immediately following the original command that branched off to the subroutine in the first place.
RIGHT$. Expressed as RIGHT$(X$,N), where X$ is some string and N is the number of letters you want counting off from the right.
RND. As in RND(X), delivers a random number from 1 to X.
ROM (Read Only Memory). Memory space unavailable to you as programmer. ROM contains the systems programs that make your computer work.
room. In an adventure, any unique area is called a room. The variable R is used to denote which room is which.
routine. In this book a routine means any chunk of a program that performs the function we happen to be talking about at the time.
RUN. The command to begin running a program.
SAVE. Command by which programs are created (or saved) on a disk.
scroll. Verb, intransitive. A term used to define how the body of text (or graphics) moves on the screen. Whenever a new line of text is added at the bottom of the already-filled screen, all the rest of the text moves up a line, while the top line disappears entirely. That is what we mean by scrolling.
semicolon (;). The symbol that suppresses carriage returns at the end of PRINT statements. Put another way, the symbol that concatenates PRINT lines.
sequential-access files. Disk files that can be accessed only from beginning to end. *Compare* direct-access files.
SOUND. As in SOUND X,Y, the command to make musical tones with your computer. X, the pitch of the tone, must be a number from 0 to 7; Y, the duration, a number from 0 to 31.
spaghetti logic. The use of innumerable GOTOs in a program to the extent that the logic is impossible to follow.
statement. A complete instruction to the computer within a program.
PRINT is a command.
10 PRINT "IT'S A BEAUTIFUL DAY IN THE NEIGHBORHOOD"
is a statement.
STEP. *See* FOR . . . NEXT.

STR$. As in STR$(X); takes the number X and turns it into a string by putting quotation marks around it.
strategy game. A game of wits for two or more players. In computer strategy games the computer takes on the role of at least one of the players.
string. In essence, as far as the computer is concerned, a string is any batch of characters between two sets of quotation marks. ''MUGWUMP'' would be a string containing the characters which are the letters that make up the word mugwump; ''123'' would also be a string, as would ''123ABC''. A series of graphics blocks is a string. I assume that the derivation of the term is from a bunch of characters being strung together.
STRING$. As in STRING$(X,Y), function that allows you to create a string containing X number of the character Y. Y can be either a string itself or an ASCII character.
structured programming. A style of programming where all the work of the processing is blocked out clearly into discrete task-performing units, making the program both intelligible and efficient. Some languages, such as Pascal, are structured languages; by definition they lead to structured programming. Others, such as Basic, have only the structure we as programmers put into them; the more structured we get, usually the better off we are.
subroutine. A self-contained portion of a program. You branch to it from within the program, get your subroutine job done, then branch back whence you came. Designing programs as a series of subroutines is considered high-class aesthetics.
syntax. The ''grammar'' of a programming statement. Correct syntax allows the line to be processed; incorrect syntax interrupts the processing.
syntax error. A mistake in your usage of Basic. Could be anything from a misspelling to misplaced punctuation, and a lot of things in between.
TAB. As in TAB(X), begins printing in the Xth column from the left.
TIME$. Every time you turn on your machine it begins tracking time. TIME$ can be set to the actual time in the real world and can also be used as a variable within programs.
TROFF. Turns off TRON, q.v.
TRON. The command that tracks each line number of a program by printing it to the screen while the program runs.
TRSDOS. The Radio Shack Disk Operating System. This is the collection of programs and commands that controls movement between the disk and the CPU.
utility. A programming aid that helps you run or create other programs.
VAL. As in VAL(any string or part thereof). Measures the numeric value of same.
variable. A symbol for a number that changes when you're not looking. Variables can consist of any combination of numbers and letters (e.g., F3GOGOMC5 is a valid variable) as long as the first character is a letter and not a number. Variables are either real number, integer, or string, and all can be used as arrays.
whistles and bells. The do-nothing parts of the program that provide nice window-dressing details.
WRITE. The command to enter data into a disk file.
Z-80. The microprocessor inside your TRS-80.

INDEX

BASIC ADVENTURE and STRATEGY GAME DESIGN for THE TRS-80®

This, the most complete and comprehensible book of its kind, strips away the mystery from game design and shows beginning-to-intermediate programmers how to create their own sophisticated computer games in TRS-80® Disk Basic.

For adventure-game designers, this book has everything you need to program your own complex adventures, including:

- how to create player puzzles
- how to monitor player and room variables
- how to develop an extensive, flexible vocabulary for conversation between the player and the computer
- *plus* a line-by-line breakdown of the original, fully playable adventure game "Space Derelict!"

For strategy-game designers, this book covers all the skills you need to design your own challenging games, including:

- how to structure the program as efficiently as possible
- how to transform strategy and tactics into computer algorithms
- how to use character graphics
- how to debug your completed program
- *plus* a line-by-line breakdown of an original, ready-to-run version of five-card draw poker that pits the human player against four computer-controlled opponents

Filled with invaluable, hard-to-find information and professional tips, and complete with glossary and index, *Basic Game Design* is the definitive guide to the subject.

Cover design by Giorgetta Bell McRee

ISBN 0-87196-977-7 >$10.95